PROMULA
FORTRAN to C Translator

User's Manual

COPYRIGHT NOTICE for
PROMULA.FORTRAN

Version 9.38 Released May, 2007

Published by:

Great Migrations LLC
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

DISCLAIMER OF WARRANTIES AND LIMITATIONS OF LIABILITIES

THIS USER'S MANUAL IS PROVIDED ON AN "AS IS" BASIS. EXCEPT FOR THE WARRANTY DESCRIBED IN THE GREAT MIGRATIONS LLC LICENSE AGREEMENT, THERE ARE NO WARRANTIES EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, AND ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED.

IN NO EVENT SHALL GREAT MIGRATIONS LLC BE RESPONSIBLE FOR ANY INDIRECT OR CONSEQUNTIAL DAMAGES OR LOST PROFITS, EVEN IF GREAT MIGRATIONS LLC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

TRADEMARK

DEFINITION OF PURCHASE

The definition of your particular purchase is specified in the Great Migrations LLC License Agreement which came with the PROMULA.FORTRAN product and which you completed and returned to Great Migrations LLC. If you have any questions about your rights or obligations as a PROMULA.FORTRAN user or believe that you have not received the complete PROMULA.FORTRAN package that you purchased, please contact:

Great Migrations LLC
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

Table Of Contents

PLEASE READ THIS SECTION **1**

1. INTRODUCTION **2**

1.1 User Support 2
1.2 What is PROMULA FORTRAN? 2
1.3 Compiler Advantages 3
1.4 Translator Advantages 3
1.5 How PROMULA FORTRAN Works 5
1.6 Rationale for Developing PROMULA FORTRAN 5
1.7 Downsizing Mainframe Codes for Use on the PC DOS Platform 6
1.8 Dealing with FORTRAN Dialect Problems 6
1.9 Dealing with C Types and FORTRAN Types 7
1.10 Dealing with FORTRAN Input/Output in C 7
1.11 Runtime Library 8
1.12 Dealing with Common Blocks 8
1.13 Allocation of Local Variables 8
1.14 A Sample Translation to C 9

2. COMMAND LINE **10**

2.1 Command Line Syntax 10
2.2 Specifying your C output bias — Bc, Bf, Bo 12
2.3 Arithmetic Conversions — CL, Cs, C0, C1, C2, C3 14
2.3.1 Arithmetic with Short Integer Variables *17*
2.4 Detailed C Output Format — CF1, CF2, CF4, CF8, CF16 20
2.5 Treatment of CHARACTER Variables — CHd, CHr, CHs, CHv 24
2.5.1 Initializing Character Values *26*
2.5.2 Subprogram Arguments *27*
2.5.3 Substrings *31*
2.5.4 Character Concatenations *32*
2.5.5 Character Treatment Conclusion *33*
2.6 Appearance of COMMENTS in C Output — CM0, CM1, CM2 33
2.7 Treatment of DATA Initializations — Da, Dc, Dr 34
2.7.1 Overview of Initialization Problem *35*
2.7.2 The Initialization Switches *37*
2.8 Turn on Debugging Mode — DB 40
2.9 Echo Control Options — ES, ET, EX, EZ, EP, EL 40
2.9.1 Warnings, Notes, and Comments *41*
2.9.2 Annotated Listing of Source Code *42*
2.9.3 Symbol Listing and Cross Reference Table *43*
2.9.4 Intermediate Compiler Tables *45*
2.9.5 Annotated Listing of C Output *46*
2.10 Treatment of Syntax Errors — ER0, ER1, ER2, ER3, ER4 46
2.11 FORTRAN Input Format Used — Fsnum, Ft, Ff, Fv, F9 48
2.12 Source FORTRAN Integer Type — FIs, FIl 49
2.13 Gname — Name of File Containing Global Symbols 49
2.14 Common Variables Convention — Ga, Gd, Gp, Gs, Gr, Gv 49
2.14.1 Overall Alignment Control with Gp — Gpc, Gps, Gpl, Gpd *53*
2.15 Iname — Name of File Containing Inline Functions 53
2.16 Target C Int Type — Is, Il 54
2.17 Treatment of Internally Generated Constants — Ka, Ks 55
2.18 Maximum Output Line Width — Lnum 57
2.19 Link Time Processing of COMMON Data Modules — Lm, Ls 57
2.20 Inclusion of Line Numbers for Debugging — Ln, L0 59
2.21 FORTRAN Dialect Selection Flags — Mdialect 60

2.22 NESTING INDENTATION TO BE USED IN THE OUTPUT — N*, N0, NN 60
2.23 INLINE COMMENTS OUTPUT MARGIN WIDTH — NCNUM 61
2.24 UPPER AND LOWER BRACES CONVENTION IN C — NU0, NU1, NU2, NL1, NL2 61
2.25 NAME OF THE FILE TO RECEIVE THE C OUTPUT — ONAME 61
2.26 SPLITTING OF OUTPUT INTO SEPARATE FILES — OS, OM 62
2.27 MISCELLANEOUS PROTOTYPING CONTROL FLAGS — PNUMB, P+NUMB 62
2.27.1 P1 — Include Definitions of int Functions 63
2.27.2 P2 — Use ANSI Prototypes for Argument Functions 63
2.27.3 P4,P8 — Exclude Referenced or Defined Prototypes 64
2.27.4 P16 — Ignore Prototypes for Definitions 65
2.27.5 P32 — Treat User Prototypes as System Functions 66
2.27.6 P64 — Write PFC Style Prototypes, not C Type 67
2.27.7 P128 — Write All Function decls to Header File 68
2.27.8 P256 — Use ANSI C Function Declarations 69
2.27.9 P512 — Make Parameters Always Take Explicit Value Type 70
2.27.10 P1024 — Exclude undefs From the Translation 71
2.27.11 P2048 — Force Variables to Have Explicit Character Type 71
2.27.12 P4096 — Define Equivalences via a #define 73
2.27.13 P8192 — Use Parameter Identifiers in Equivalences 74
2.27.14 P16384 — Display Include Files Separately 75
2.28 LISTING FILE CONTROL — PANAME, PHNUMB, PNNAME, PWNUMB 77
2.29 QUANTITY CONTROL FLAGS — QINUMB, QENUMB, QDNUMB, QXNUMB, QHNUMB, QWNUMB 77
2.29.1 QInumb — Size of Compacted Statement Storage 78
2.29.2 QEnumb — Size of the Line Number Table 78
2.29.3 QDnumb — Size of a Data Block 78
2.29.4 QXnumb — Size of External Information Storage 78
2.29.5 QHnumb — Size of Include File Information Storage 79
2.29.6 QWnumb — Word Size of Source Platform 79
2.30 SPECIFY A CONFIGURATION FILE — RNAME 79
2.31 STORAGE THRESHOLD VALUES — SANUM, SDNUM, SSNUM, SVNUM, SZNUM 79
2.32 FORTRAN DIALECT DOLOOP ASSUMPTIONS — T0, T1, T2 83
2.33 TREATMENT OF INTERNALLY GENERATED TEMPORARIES — TA, TS 87
2.34 SPECIFYING UNIT NUMBERS — UR, URNUM, UP, UPNUM, UW, UWNUM 89
2.35 FILE TO RECEIVE PROTOTYPE DEFINITIONS — WNAME 89
2.36 MISCELLANEOUS CONTROL FLAGS — Y1, Y2 89
2.36.1 The Treatment of Entry Points — Y1 90
2.36.2 Output Form of Parameter Identifiers — Y2 92
2.37 TREATMENT OF MULTIPLE ASSIGNMENTS — XA, YA 92
2.38 TREATMENT OF SINGLE STATEMENT NESTING BRACE — XB, YB 93
2.39 CONSTANT REDUCTION OPTIMIZATION — XC, YC 94
2.40 CHARACTER OPTIMIZATION SWITCHES — XCH, YCH 96
2.41 TREATMENT OF FORTRAN "D" DEBUGGING STATEMENTS 96
2.41.1 Treatment of Other Debugging Statements — Ydstring 97
2.42 USE OF PRINTF-STYLE FORMATTING — XF, YF 98
2.43 INITIALIZATION CHECK FOR AUTO VARIABLES — XI, YI 99
2.44 DO LOOP COUNTER REDUCTION OPTIMIZATION — XL, YL 99
2.45 SUBPROGRAM ARGUMENT TYPE CHECKING — XP, YP 101
2.46 SINGLE PRECISION REAL ARITHMETIC — XR, YR 101
2.47 SUBSCRIPT POINTER NOTATION — XS, YS, YSV, YSF 101
2.48 UNFORMATTED WRITE OPTIMIZATION — XU, YU 104
2.49 SUBPROGRAM CALL-BY-VALUE ARGUMENTS — XV, YV 105
2.50 DOLLAR SIGNS AS INITIAL SYMBOLS IN IDENTIFIERS — X$, Y$ 105
2.51 LOCATION OF FORTRAN FILES TO BE INCLUDED — ZNAME 105
2.52 PROJECT PROCESSING — #PROJECT 105

3. CONFIGURATION FILE 107

3.1 THE CONFIGURATION SWITCHES STATEMENT 107

3.2 THE CONFIGURATION COMMENTS STATEMENT 108
3.3 THE CONFIGURATION PATHNAMES STATEMENT 112
3.4 THE CONFIGURATION RESTRUCTURE STATEMENT 114
3.5 THE CONFIGURATION KEYWORDS STATEMENT 116
3.5.1 Simple Keyword Replacement 119
3.5.2 Pattern Strings for COMMON blocks 120
3.5.3 Pattern String for External Functions 121
3.5.4 Pattern String for Subprogram Surrogates 122
3.5.5 Pattern Strings for VAX Descriptors 123
3.6 THE CONFIGURATION PRAGMA STATEMENT 123
3.7 THE CONFIGURATION $ STATEMENT 125

4. THE CONFIGURATION FUNCTION PROTOTYPES 127

4.1 FUNCTION PROTOTYPE SYNTAX 127
4.2 VALUE PARAMETERS 128
4.3 EXTERNAL NAME CLASH 129
4.4 MULTIPLE FORMS 130
4.5 GLOBAL SYMBOLS AND PROTOTYPES 132
4.6 RENAMING IDENTIFIERS ONLY 132

5. OVERVIEW OF RUNTIME LIBRARY 133

5.1 NAMING AND ORGANIZATION OF FUNCTIONS 133
5.2 GENERAL FORTRAN OPERATIONS 134
5.3 INPUT/OUTPUT OPERATIONS 134
5.3.1 Runtime Error Messages 134
5.4 NONCOMPLEX INTRINSIC FUNCTIONS 135
5.5 VIRTUAL MEMORY SYSTEM 136
5.5.1 The Virtual Memory Management Algorithm 136
5.5.2 Virtual Memory Global Variables 137
5.6 SINGLE PRECISION COMPLEX ARITHMETIC 138
5.7 DOUBLE PRECISION COMPLEX ARITHMETIC 138

6. RUNTIME LIBRARY FUNCTION DESCRIPTIONS 139

6.1 CPXABS: COMPUTE THE SHORT COMPLEX ABSOLUTE VALUE 139
6.2 CPXADD: SHORT COMPLEX ADDITION 139
6.3 CPXCJG: COMPUTE THE SHORT COMPLEX CONJUGATE 140
6.4 CPXCMP: SHORT COMPLEX COMPARISON 140
6.5 CPXCOS: COMPUTE THE SHORT COMPLEX COSINE 141
6.6 CPXCPX: CONVERT TWO FLOATS TO SHORT COMPLEX 141
6.7 CPXDBL: CONVERT DOUBLE PRECISION TO SHORT COMPLEX 142
6.8 CPXDIV: SHORT COMPLEX DIVISION 142
6.9 CPXDPX: CONVERT DOUBLE COMPLEX TO SHORT COMPLEX 142
6.10 CPXEXP: SHORT COMPLEX EXPONENTIAL 143
6.11 CPXIMA: COMPUTE THE IMAGINARY PART OF A SHORT COMPLEX 143
6.12 CPXLOG: SHORT COMPLEX NATURAL LOGARITHM 144
6.13 CPXLOG10: SHORT COMPLEX BASE 10 LOGARITHM 144
6.14 CPXLONG: CONVERT SHORT COMPLEX TO LONG 145
6.15 CPXMUL: SHORT COMPLEX MULTIPLICATION 145
6.16 CPXNEG: COMPUTE THE SHORT COMPLEX NEGATIVE 146
6.17 CPXPOL: SHORT COMPLEX CONVERSION TO POLAR 146
6.18 CPXPOW: RAISE SHORT COMPLEX TO A POWER 147
6.19 CPXREAL: COMPUTE REAL PART OF SHORT COMPLEX 147
6.20 CPXSIN: COMPUTE THE SHORT COMPLEX SINE 147
6.21 CPXSROOT: COMPUTE SHORT COMPLEX SQUARE ROOT 148
6.22 CPXSUB: SHORT COMPLEX SUBTRACTION 148
6.23 DPXABS: COMPUTE THE DOUBLE COMPLEX ABSOLUTE VALUE 149
6.24 DPXADD: DOUBLE COMPLEX ADDITION 149

6.25 DPXCJG: COMPUTE THE DOUBLE COMPLEX CONJUGATE 150
6.26 DPXCMP: DOUBLE COMPLEX COMPARISON 150
6.27 DPXCOS: COMPUTE THE DOUBLE COMPLEX COSINE 151
6.28 DPXCPX: CONVERT SHORT COMPLEX TO DOUBLE COMPLEX 151
6.29 DPXDBL: CONVERT DOUBLE PRECISION TO DOUBLE COMPLEX 152
6.30 DPXDIV: DOUBLE COMPLEX DIVISION 152
6.31 DPXDPX: CONVERT TWO DOUBLES TO DOUBLE COMPLEX 153
6.32 DPXEXP: DOUBLE COMPLEX EXPONENTIAL 153
6.33 DPXIMA: COMPUTE IMAGINARY OF DOUBLE COMPLEX 154
6.34 DPXLOG: DOUBLE COMPLEX NATURAL LOGARITHM 154
6.35 DPXLOG10: DOUBLE COMPLEX BASE 10 LOGARITHM 155
6.36 DPXLONG: CONVERT DOUBLE COMPLEX TO LONG 155
6.37 DPXMUL: DOUBLE COMPLEX MULTIPLICATION 155
6.38 DPXNEG: COMPUTE THE DOUBLE COMPLEX NEGATIVE 156
6.39 DPXPOL: DOUBLE COMPLEX CONVERSION TO POLAR 156
6.40 DPXPOW: RAISE DOUBLE COMPLEX TO A POWER 157
6.41 DPXREAL: COMPUTE REAL PART OF DOUBLE COMPLEX 157
6.42 DPXSIN: COMPUTE THE DOUBLE COMPLEX SINE 158
6.43 DPXSROOT: COMPUTE DOUBLE COMPLEX SQUARE ROOT 158
6.44 DPXSUB: DOUBLE COMPLEX SUBTRACTION 159
6.45 FIFAMAX0: FORTRAN INTRINSIC FUNCTION AMAX0 159
6.46 FIFAMIN0: FORTRAN INTRINSIC FUNCTION AMIN0 160
6.47 FIFASC50: FORTRAN EXTERNAL FUNCTION ASC50 160
6.48 FIFCHAR: FORTRAN INTRINSIC FUNCTION CHAR 161
6.49 FIFCOS: FORTRAN INTRINSIC FUNCTION COS 161
6.50 FIFDATE: EXTERNAL FUNCTION DATA 161
6.51 FIFDDIM: FORTRAN INTRINSIC FUNCTION DDIM 162
6.52 FIFDINT: FORTRAN INTRINSIC FUNCTION DINT 162
6.53 FIFDMAX1: FORTRAN INTRINSIC FUNCTION DMAX1 163
6.54 FIFDMIN1: FORTRAN INTRINSIC FUNCTION DMIN1 163
6.55 FIFDMOD: FORTRAN INTRINSIC FUNCTION 163
6.56 FIFDNINT: FORTRAN INTRINSIC FUNCTION DNINT 164
6.57 FIFDSIGN: FORTRAN INTRINSIC FUNCTION DSIGN 164
6.58 FIFEQF: FORTRAN INTRINSIC FUNCTION EQF 165
6.59 FIFEXIT: FORTRAN EXIT SUBROUTINE 165
6.60 FIFGETAR: FORTRAN GET COMMAND LINE ARGUMENTS 165
6.61 FIFGETCL: FORTRAN GET COMMAND LINE SUBROUTINE 166
6.62 FIFGETENV: FORTRAN GET VALUE OF ENVIRONMENT VARIABLES 166
6.63 FIFHBIT: FORTRAN HIGH BIT MANAGEMENT 167
6.64 FIFI2ABS: FORTRAN INTRINSIC FUNCTION I2ABS 167
6.65 FIFI2DAT: FORTRAN EXTERNAL FUNCTION I2DATE 167
6.66 FIFI2DIM: FORTRAN INTRINSIC FUNCTION I2DIM 168
6.67 FIFI2DINT: FORTRAN INTRINSIC FUNCTION I2DINT 168
6.68 FIFI2MAX0: FORTRAN INTRINSIC FUNCTION I2MAX0 168
6.69 FIFI2MIN0: FORTRAN INTRINSIC FUNCTION I2MIN0 169
6.70 FIFI2MOD: FORTRAN INTRINSIC FUNCTION I2MOD 169
6.71 FIFI2NINT: FORTRAN INTRINSIC FUNCTION I2NINT 170
6.72 FIFI2POW: FORTRAN INTRINSIC FUNCTION I2POW 170
6.73 FIFI2SHF: FORTRAN INTRINSIC FUNCTION I2SHFT 170
6.74 FIFI2SIGN: FORTRAN INTRINSIC FUNCTION I2SIGN 171
6.75 FIFIABS: FORTRAN INTRINSIC FUNCTION IABS 171
6.76 FIFIARGC: FORTRAN GET COMMAND LINE ARGUMENT COUNT 171
6.77 FIFIBIT: FORTRAN INTRINSIC FUNCTION IBIT 172
6.78 FIFICHAR: FORTRAN INTRINSIC FUNCTION ICHAR 172
6.79 FIFIDIM: FORTRAN INTRINSIC FUNCTION IDIM 173
6.80 FIFIDINT: FORTRAN INTRINSIC FUNCTION IDINT 173
6.81 FIFINDEX: FORTRAN INTRINSIC FUNCTION INDEX 173

6.82 FIFIPOW: FORTRAN INTRINSIC FUNCTION IPOW 174
6.83 FIFISHF: FORTRAN INTRINSIC FUNCTION ISHFT 174
6.84 FIFISIGN: FORTRAN INTRINSIC FUNCTION ISIGN 175
6.85 FIFMAX0: FORTRAN INTRINSIC FUNCTION MAX0 175
6.86 FIFMAX1: FORTRAN INTRINSIC FUNCTION MAX1 175
6.87 FIFMIN0: FORTRAN INTRINSIC FUNCTION MIN0 176
6.88 FIFMIN1: FORTRAN INTRINSIC FUNCTION MIN1 176
6.89 FIFMOD: FORTRAN INTRINSIC FUNCTION MOD 176
6.90 FIFNEF: FORTRAN INTRINSIC FUNCTION NEF 177
6.91 FIFNINT: FORTRAN INTRINSIC FUNCTION NINT 177
6.92 FIFRAD50: FORTRAN EXTERNAL FUNCTION IRAD50 178
6.93 FIFRBIT: FORTRAN INTRINSIC FUNCTION RBIT 178
6.94 FIFSIN: FORTRAN INTRINSIC FUNCTION SIN 179
6.95 FIFSNCS: FORTRAN SINGLE PRECISION SINE/COSINE 179
6.96 FIFSTRGV: FORTRAN STRING VALUE CONVERSION 179
6.97 FIFSYSTM: FORTRAN EXTERNAL FUNCTION SYSTEM 180
6.98 FIFTAN: FORTRAN INTRINSIC FUNCTION TAN 180
6.99 FIFTIME: FORTRAN EXTERNAL FUNCTION TIME 181
6.100 FIFXBIT: FORTRAN INTRINSIC FUNCTION XBIT 181
6.101 FIFXCREP: FORTRAN EXTENDED CHARACTER REPRESENTATION 181
6.102 FIOBACK: BACKSPACE A FORTRAN FILE 182
6.103 FIOBFOUT: BUSINESS FORMAT OUTPUT 182
6.104 FIOCLOSE: CLOSE CURRENT FORTRAN FILE 184
6.105 FIOCPATH: CONVERT PATHNAME 184
6.106 FIODTOS: CONVERT DOUBLE VALUE TO STRING 185
6.107 FIOERROR: PERFORM FORTRAN I/O ERROR PROCESSING 185
6.108 FIOFDATA: FORTRAN FILE DATA 186
6.109 FIOFEND: END FORMAT PROCESSING 187
6.110 FIOFFLD: GET NEXT FREE-FORM FIELD 188
6.111 FIOFINI: INITIALIZE A FORTRAN FORMAT 188
6.112 FIOFINP: FORMATTED INPUT 189
6.113 FIOFINQU: INQUIRE ABOUT FILE DATA 190
6.114 FIOFMTV: COMPUTE FORMAT VALUE 191
6.115 FIOFOUT: FORMATTED OUTPUT OPERATIONS 191
6.116 FIOFVINQ: INQUIRE ABOUT FILE VALUE 192
6.117 FIOFWSP: SKIP FORMAT WHITE SPACE 193
6.118 FIOINTU: ESTABLISH FORTRAN INTERNAL UNIT 193
6.119 FIOITOS: CONVERT INTEGER TO STRING 194
6.120 FIOLREC: POSITION A FORTRAN FILE ON A RECORD 194
6.121 FIOLTOS: CONVERT LONG INTEGER TO STRING 195
6.122 FIOLUN: ESTABLISH FORTRAN UNIT NUMBER 195
6.123 FIONAME: ESTABLISH FORTRAN UNIT BY NAME 196
6.124 FIONXTF: GET NEXT FORMAT SPECIFICATION 196
6.125 FIOOPEN: OPEN CURRENT FORTRAN FILE 197
6.126 FIORALPH: READ ALPHABETIC INFORMATION 198
6.127 FIORBIV: FORTRAN READ BINARY VALUES 198
6.128 FIORCHK: CHECK FIXED-FORM INPUT FIELD 199
6.129 FIORDB: READ FORTRAN BOOLEAN VECTOR 199
6.130 FIORDC: READ FORTRAN CHARACTER VECTOR 200
6.131 FIORDD: READ FORTRAN DOUBLE PRECISION VECTOR 200
6.132 FIORDF: READ FORTRAN FLOATING POINT VALUES 201
6.133 FIORDI: READ FORTRAN SHORT INTEGER VECTOR 201
6.134 FIORDL: READ FORTRAN LONG INTEGER VECTOR 202
6.135 FIORDS: READ FORTRAN STRING 202
6.136 FIORDT: READ FORTRAN TRUTH VALUE VECTOR 203
6.137 FIORDU: READ FORTRAN UNSIGNED CHAR VECTOR 203
6.138 FIORDX: READ FORTRAN COMPLEX VALUES 204

6.139 FIORDZ: READ FORTRAN DOUBLE COMPLEX VALUES 204
6.140 FIOREC: POSITION A FORTRAN FILE ON A RECORD 205
6.141 FIOREW: REWIND A FORTRAN FILE 205
6.142 FIORLN: READ FORTRAN END-OF-LINE 206
6.143 FIORNDV: ROUND VALUE 206
6.144 FIORNL: PROCESS FORTRAN READ DATALIST STATEMENT 207
6.145 FIORPATH: READ PATHNAME CONVERSION INFORMATION 207
6.146 FIORTXT: READ NEXT TEXT RECORD 209
6.147 FIORWBV: FORTRAN REWRITE BINARY VALUES 210
6.148 FIOSHL: SHIFT STRING LEFT 210
6.149 FIOSHR: SHIFT STRING RIGHT 210
6.150 FIOSPACE: SKIP WHITE SPACE IN RECORD 211
6.151 FIOSTATUS: SET FORTRAN I/O ERROR STATUS 211
6.152 FIOSTIO: ESTABLISH FORTRAN STANDARD I/O 211
6.153 FIOSTOD: CONVERT STRING TO DOUBLE 212
6.154 FIOSTOI: CONVERT STRING TO INTEGER 213
6.155 FIOUWL: ESTABLISH FORTRAN UNFORMATTED WRITE LENGTH 213
6.156 FIOVFINI: INITIALIZE A VARIABLE FORTRAN FORMAT 213
6.157 FIOWALPH: WRITE ALPHABETIC INFORMATION 214
6.158 FIOWBIV: FORTRAN WRITE BINARY VALUES 214
6.159 FIOWDBL: WRITE DOUBLE PRECISION VALUE 215
6.160 FIOWEF: FORTRAN WRITE END-OF-FILE 215
6.161 FIOWHEXO: WRITE HEXADECIMAL OR OCTAL CONSTANT 216
6.162 FIOWLN: WRITE FORTRAN END-OF-LINE 216
6.163 FIOWNL: PROCESS FORTRAN WRITE DATALIST STATEMENT 216
6.164 FIOWRB: WRITE FORTRAN BOOLEAN VECTOR 217
6.165 FIOWRC: WRITE FORTRAN CHARACTER VECTOR 218
6.166 FIOWRD: WRITE FORTRAN DOUBLE PRECISION VECTOR 218
6.167 FIOWRF: WRITE FORTRAN SINGLE PRECISION VECTOR 219
6.168 FIOWRI: WRITE FORTRAN SHORT INTEGER VECTOR 219
6.169 FIOWRL: WRITE FORTRAN LONG INTEGER VECTOR 220
6.170 FIOWRS: WRITE FORTRAN VECTOR OF STRINGS 220
6.171 FIOWRT: WRITE FORTRAN TRUTH VALUE VECTOR 221
6.172 FIOWRU: WRITE FORTRAN UNSIGNED CHAR VECTOR 221
6.173 FIOWRX: WRITE FORTRAN COMPLEX VECTOR 222
6.174 FIOWTXT: WRITE TEXT RECORD 222
6.175 FIOWVAL: WRITE FLOATING POINT VALUE 223
6.176 FIOWVB: WRITE FORTRAN BOOLEAN VALUE 224
6.177 FIOWVC: WRITE FORTRAN CHARACTER VALUE 224
6.178 FIOWVD: WRITE FORTRAN DOUBLE VALUE 224
6.179 FIOWVF: WRITE FORTRAN FLOAT VALUE 225
6.180 FIOWVI: WRITE FORTRAN SHORT INTEGER VALUE 225
6.181 FIOWVL: WRITE FORTRAN LONG INTEGER VALUE 226
6.182 FIOWVS: WRITE FORTRAN STRING VALUE 226
6.183 FIOWVT: WRITE FORTRAN TRUTH VALUE 227
6.184 FIOWVU: WRITE FORTRAN CHARACTER VALUE 227
6.185 FIOWVX: WRITE FORTRAN COMPLEX VALUE 228
6.186 FIOWVZ: WRITE FORTRAN DOUBLE COMPLEX VALUE 228
6.187 FTNADS: FORTRAN ADD STRINGS 229
6.188 FTNALLOC: ALLOCATE DYNAMIC MEMORY 229
6.189 FTNBACK: FORTRAN BACKSPACE STATEMENT 230
6.190 FTNBLKD: FORTRAN BLOCK DATA 230
6.191 FTNCLOSE: FORTRAN CLOSE STATEMENT 230
6.192 FTNCMS: FORTRAN COMPARE STRINGS 231
6.193 FTNFREE: FREE DYNAMIC MEMORY 232
6.194 FTNINI: INITIALIZE FORTRAN PROCESSING 232
6.195 FTNLUN: ESTABLISH FILE FOR LOGICAL UNIT NUMBER 233

6.196 FTNOPEN: FORTRAN OPEN STATEMENT 233
6.197 FTNPAUSE: FORTRAN PAUSE STATEMENT 234
6.198 FTNREAD: FORTRAN READ STATEMENT 234
6.199 FTNREW: FORTRAN REWIND STATEMENT 235
6.200 FTNSAC: FORTRAN STORE A CHARACTER STRING 236
6.201 FTNSALLO: FORTRAN STRING ALLOCATION 237
6.202 FTNSCOMP: FORTRAN STRING COMPARISON 237
6.203 FTNSCOPY: FORTRAN STRING COPY 237
6.204 FTNSLENG: FORTRAN STRING LENGTH 238
6.205 FTNSUBS: FORTRAN SUBSTRING EVALUATION 238
6.206 FTNSTOP: FORTRAN STOP STATEMENT 239
6.207 FTNWEF: FORTRAN END FILE STATEMENT 239
6.208 FTNWRIT: FORTRAN WRITE STATEMENT 240
6.209 FTNXCONS: FORTRAN EXACT REPRESENTATION CONSTANT 241
6.210 P77GETU: PRIME FORTRAN 77 FUNCTION F77$GETU 242
6.211 P77NLENA: PRIME FORTRAN 77 SUBROUTINE NLEN$A 242
6.212 P77TNOUA: PRIME FORTRAN 77 SUBROUTINE TNOUA 242
6.213 PDPASSN: PDP FORTRAN SUBROUTINE ASSIGN 243
6.214 PDPCLOSE: PDP FORTRAN SUBROUTINE CLOSE 243
6.215 PDPCVTIM: PDP FORTRAN EXTERNAL FUNCTION CVTTIM 244
6.216 PDPGTIM: PDP FORTRAN EXTERNAL FUNCTION GTIM 244
6.217 VMSCLS: CLOSE VIRTUAL FILE 245
6.218 VMSDEL: CHANGE VIRTUAL INFORMATION 245
6.219 VMSGLOB: VIRTUAL GLOBAL ACCESS 246
6.220 VMSLOAD: LOAD A VIRTUAL VECTOR 246
6.221 VMSOPN: OPEN A VIRTUAL MEMORY FILE 246
6.222 VMSPTR: GET VIRTUAL BYTE POINTER 247
6.223 VMSRBL: REMOVE VIRTUAL BLOCK 248
6.224 VMSRDB: READ FORTRAN VIRTUAL BOOLEAN VECTOR 248
6.225 VMSRDC: READ FORTRAN VIRTUAL CHARACTER VECTOR 249
6.226 VMSRDD: READ FORTRAN VIRTUAL DOUBLE PRECISION VECTOR 249
6.227 VMSRDF: READ FORTRAN VIRTUAL FLOATING POINT VALUES 250
6.228 VMSRDI: READ FORTRAN VIRTUAL SHORT INTEGER VECTOR 250
6.229 VMSRDL: READ FORTRAN VIRTUAL LONG INTEGER VECTOR 251
6.230 VMSRDS: READ FORTRAN VIRTUAL STRING 251
6.231 VMSRDT: READ FORTRAN VIRTUAL TRUTH-VALUE VECTOR 252
6.232 VMSRDU: READ FORTRAN VIRTUAL UNSIGNED CHARACTER VECTOR 252
6.233 VMSSAVE: SAVE A VIRTUAL VECTOR 253
6.234 VMSUSE: USE VIRTUAL INFORMATION 253
6.235 VMSVECT: VIRTUAL VECTOR INPUT/OUTPUT 254
6.236 VMSWRB: WRITE FORTRAN VIRTUAL BOOLEAN VECTOR 254
6.237 VMSWRC: WRITE FORTRAN VIRTUAL CHARACTER VECTOR 255
6.238 VMSWRD: WRITE FORTRAN VIRTUAL DOUBLE PRECISION VECTOR 255
6.239 VMSWRF: WRITE FORTRAN VIRTUAL SINGLE PRECISION VECTOR 256
6.240 VMSWRI: WRITE FORTRAN VIRTUAL SHORT INTEGER VECTOR 256
6.241 VMSWRL: WRITE FORTRAN VIRTUAL LONG INTEGER VECTOR 257
6.242 VMSWRS: WRITE FORTRAN VIRTUAL VECTOR OF STRINGS 257
6.243 VMSWRT: WRITE A VIRTUAL LONG TRUTH VALUE VECTOR 258
6.244 VMSWRU: WRITE FORTRAN VIRTUAL UNSIGNED CHARACTER VECTOR 258
6.245 VMSWVB: WRITE A VIRTUAL BLOCK 259

PLEASE READ THIS SECTION

There are three fundamentally different ways of using PFC:

1. As a FORTRAN compiler;

2. As a tool to produce maintainable C source code from a FORTRAN source which corresponds to the original as closely as possible so that it can be maintained by the original authors;

3. As a tool to produce a maintainable C source code from a FORTRAN source which is logically equivalent to the original, but which uses conventional C notation and standard C functions as much as possible.

These three views are referred to as the "optimized", "FORTRAN", and "C" biases respectively. The default bias for PFC is the FORTRAN bias. The C and optimized biases are, however, fully supported by PFC and may be activated by the Bc and Bo command line switches. In addition, the PFC configuration file can be easily changed to make either the C or the optimized bias the default.

Please evaluate your own reasons for applying PFC to your FORTRAN codes and make the appropriate bias selection. If you are still not satisfied with the output from PFC, please look at the chapters in this manual which discuss the command line switches available and the use of a configuration file. Virtually every aspect of the look of the output can easily be controlled by you. If you are still not satisfied, please contact our user support staff. We feel very strongly that PFC can support any reasonable objective or bias, and would like the opportunity to prove it.

1. INTRODUCTION

The PROMULA FORTRAN to C Translator generates C code which can be compiled by a standard C compiler to produce executable code. It is operational on a wide variety of platforms. More than just a translator, PROMULA FORTRAN adds value to the code during the translation: virtual memory logic, dynamic memory logic, references to external databases and application management systems, and integration with other operating environments. Finally, PROMULA FORTRAN has a dialect management component which allows the customization of the package in terms of both the FORTRAN dialect to be accepted by the translator and the form of the C translation output.

The source code processing component of PROMULA.FORTRAN is completely compatible with the one used by the PROMULA FORTRAN Compiler. The manual for the PROMULA FORTRAN Compiler is included with this manual. That manual contains a description of the FORTRAN language supported, controlling runtime behavior, the PROMULA interface, and error messages. That discussion applies directly to PROMULA.FORTRAN as well and will not be repeated in this manual.

1.1 User Support

If you are a licensed and registered user of PROMULA FORTRAN, you are entitled to user support from Great Migrations LLC.

If you encounter a problem that you cannot resolve on your own by referring to this User's Manual, you may call or write us:

Great Migrations LLC
PFC Support
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

Your comments and suggestions about the product are always welcome.

If possible, we will provide help over the telephone. However, if the problem involves an apparent translation problem or a runtime library problem, we will probably need a copy of your source FORTRAN. We will protect the full confidentiality of any sample codes that you send to us.

If the problem does uncover a problem either with the translator or with the runtime library, you will be supplied with a corrected copy of PROMULA FORTRAN and/or the Runtime Library as soon as we have made those corrections.

All purchasers of PROMULA FORTRAN will be notified of any revised versions, and will be given the option to purchase them at a nominal update cost.

At your request, Great Migrations LLC staff will also provide technical consulting services to assist you in software conversion projects.

1.2 What is PROMULA FORTRAN?

PROMULA FORTRAN is a compiler which will process FORTRAN codes of almost any dialect on almost any platform that supports a standard C compiler. It is also a comprehensive FORTRAN to C translator which converts FORTRAN code to clean, portable, and maintainable C code while allowing extensive control over the translation process.

No matter how old or how extended your FORTRAN dialect is, PROMULA will process it by first compiling it to the more versatile and more portable C language. Your long-established FORTRAN programs do not have to be maintenance burdens running inefficiently on old platforms; with PROMULA, you can give them new life on contemporary platforms where you can take advantage of new technology options, including the option of program maintenance in either FORTRAN or C.

1.3 Compiler Advantages

As a FORTRAN compiler, PROMULA FORTRAN offers a number of advantages over other FORTRAN compilers:

Portability — Compile to C rather than machine code. Maintain a single FORTRAN source code on multiple platforms. Port and process your applications on almost any platform that supports a standard C compiler.

Multi-Dialect Processing — Compile standard FORTRAN 66 and FORTRAN 77 dialects as well as various other extended dialects, such as VAX, IBM VS, PDP, PRIME, Honeywell, and Data General FORTRAN.

Multi-Platform Availability — Achieve reproducible results on multiple platforms without having to maintain separate source codes on each platform. When you migrate from one platform to another, bring your FORTRAN applications with you — including your FORTRAN compiler.

PROMULA FORTRAN is available for several platforms: IBM PC, Apple Macintosh, VAX/VMS, VAX/ULTRIX, SUN/UNIX, IBM/AIX, 386/UNIX and other UNIX workstations, as well as IBM mainframes. PROMULA FORTRAN is a portable C program and can be installed via shrouded source code on platforms not listed above, provided they support a standard C compiler.

Run-Time Library — PROMULA FORTRAN comes with an extensive runtime library which reproduces in C the full functionality of FORTRAN (FORTRAN I/O, complex arithmetic, etc.). The library is available in C source code and can be recompiled with any standard C compiler. This means that you can use PROMULA FORTRAN as a cross-compiler, i.e., translate your FORTRAN code to C in one platform using PROMULA FORTRAN, then compile, link and run the translated C code on a second platform using a C compiler.

Validation — The compiler has been tested on several platforms with Version 2.0 of the FORTRAN compiler Validation System from the Federal Software Testing Center and passed the test at the full validation level on all platforms.

Integration — Upgrade your FORTRAN applications by integrating them naturally with GUI libraries and other C-based software on new platforms.

Debugging — Extensive error checking is done at three steps of the process:

(a) During compilation of the FORTRAN to C, by PROMULA FORTRAN.

(b) During compilation of the translated C code, by the C compiler. In principle, after a successful Step (a) above, this step yields no compilation errors.

(c) During execution, by the debugger of the C compiler. For UNIX-based systems, the dbx debugger may be used and it can reference either the translated C code or the lines of the original FORTRAN source code.

1.4 Translator Advantages

As a FORTRAN to C translator, PROMULA FORTRAN offers a number of advantages over other FORTRAN to C translators:

Multi-Dialect Processing Translate the standard FORTRAN 66 and FORTRAN 77 dialects as well as various other extended dialects, such as VAX, IBM VS, PDP, PRIME, Honeywell, and Data General FORTRAN. A particular FORTRAN dialect is selected at the time of purchase. Additional FORTRAN dialects may be purchased later.

Cross-Platform Availability Translate your code on one platform for compilation, linking and running on a second platform.

Completeness The syntax processor handles almost all existing standard and extended FORTRAN dialects. It also supports a number of Fortran 90 features, such as structures and pointer variables. On the execution side, its comprehensive runtime library covers the full functionality of all of the above FORTRAN dialects (full FORTRAN I/O, complex arithmetic, NAMELIST, DECODE/ENCODE, DEFINE FILE, B-FORMAT, %VAL, %REF, %DESCR, STRUCTURE, RECORD, embedded and inline comments, etc.)

Correctness Since PROMULA FORTRAN compiles to C, i.e., performs both a syntactic and a semantic transformation to C, the translated C code is both compilable and "correct," that is, it yields reproducible results. See Validation above.

Language Migration Translate FORTRAN to maintainable C. Migrate to C for source code maintenance and further development.

The C code generator actually offers three output options (or biases):

(a) The FORTRAN bias generates C output which is as close to the original FORTRAN as possible and is aimed at easing the transition of those users who are presently FORTRAN programmers but wish to (or must) become C programmers.

(b) The C bias generates C output which looks much like a standard C program and is aimed at those users who are C programmers but must now take over the maintenance of a FORTRAN code.

(c) The optimized bias generates C output which is designed to compile as quickly as possible and to produce an efficient as possible executable module. This output is not very readable and is aimed at those users who wish to continue to program in FORTRAN. For these users, the C output is of no importance as such. It is merely an intermediate step and serves as input to the C compiler.

Error Processing When a FORTRAN error occurs, an error message is issued and the user has the option to exit or continue processing. A command-line switch allows the user to select from five different error handling options:

(a) Stop translation when the first error is encountered

(b) Translate all errors into non-compilable FORTRAN ERROR statements

(c) Translate all errors into non-linkable FORTRAN ERROR function calls

(d) Translate all errors into executable print-error-message statements

(e) Translate all errors into warning comments.

Conversion Support As part of our standard maintenance agreement, if after translating a working FORTRAN application with PROMULA FORTRAN you find that it does not yield the same results in C as it does in FORTRAN, we will gladly revise or extend our translator and/or its runtime library until you achieve complete reproducibility of results. This guarantee applies only to pure FORTRAN applications; hybrid applications that contain external, possibly non-FORTRAN, components, are outside the scope of our standard maintenance agreement and are handled as part of our system conversion services on a consulting basis, upon request.

Documentation According to a *Computer Language* review (October, 1988), "the documentation for PROMULA FORTRAN is excellent. It meticulously details the translation process,... all the runtime library routines, and more." In addition, the language reference component of the documentation describes the actual FORTRAN dialect that PROMULA FORTRAN supports.

1.5 How PROMULA FORTRAN Works

The PROMULA compiler is based on the proven FORTRAN to C translator which was released to the PC market almost six years ago as Version 1.0 of PROMULA FORTRAN and has shown in a number of FORTRAN code migration projects that reproducible results can be achieved almost automatically without lengthy and expensive manual recoding work.

PROMULA FORTRAN translates FORTRAN code to C code which is then compiled via any standard C compiler and linked with the PROMULA FORTRAN runtime library to produce efficient executable code. The resultant executable code produces the same results on a target platform as the original code does on the source platform.

In designing PROMULA FORTRAN we took the position that the only difference between a translator and a compiler should be that a compiler converts the source code into machine language while a translator takes it to a higher level language. PROMULA FORTRAN compiles the FORTRAN source language into a low level pseudocode. This pseudo-code is much like the output produced by the first, or second, pass of contemporary compilers. Second, it optimizes that code using the same techniques as used during the optimization pass of a compiler. Third, it does code generation; but the code generated is not machine code, it is C.

A more detailed description of the design and methodology of PROMULA FORTRAN appeared in a series of three technical papers in the *Journal of C Language Translation*:

1. "Design of a FORTRAN to C Translator," Fred K. Goodman, Vol. 1, December, 1989 and March, 1990.

2. "FORTRAN to C: Numerical Issues," Fred K. Goodman, Vol. 2, June, 1990.

3. "FORTRAN to C: Character Manipulation," Fred K. Goodman, Vol. 2, September, 1990.

1.6 Rationale for Developing PROMULA FORTRAN

Since the beginning of FORTRAN there has always been a problem — FORTRAN is not transportable from machine to machine or even from one operating system to another.

In contrast, the C language is unique in that it is available for almost every type of computer — from home computers to supercomputers. It is extremely efficient, modular, and portable. It is presently the language of choice for many operating system programmers and compiler designers.

In addition to authoring PROMULA FORTRAN, we have been developing FORTRAN development tools and FORTRAN applications for clients since 1967. In converting FORTRAN programs from one platform to another, our typical problem was not that we could not find a good FORTRAN 77 compiler for the target platform, but rather that we were confronted by FORTRAN programs that were almost always written in non-standard, non-machine-transportable FORTRAN dialects. The typical mainframe FORTRAN program is not written in standard FORTRAN and makes assumptions about the machine and operating system for which it was originally written. FORTRAN programs are just not portable.

PROMULA FORTRAN was designed to deal with actual FORTRAN programs, written by "non- structured" FORTRAN programmers who took advantage of every possible special feature of their particular vendor's compiler, and who had never known or cared that there was a standard, or two, available.

If you have a relatively small FORTRAN 77 program and wish to use it on a new platform in a C environment, then you have a variety of compilers and/or translators available to you, but we think PROMULA FORTRAN will give you the best results. However, if you have a serious FORTRAN program and you do not wish to do any changes by hand, then we know that PROMULA FORTRAN is your only current alternative.

We originally developed PROMULA FORTRAN for use in our own consulting business because we were unable to find a FORTRAN compiler which would effectively and accurately process the typical FORTRAN programs which we wished to migrate to the PC or other workstations. Having once developed the translator, we discovered that far more than just translation could be achieved during the conversion. Many other problems could also be solved by translating to C.

1.7 Downsizing Mainframe Codes for Use on the PC DOS Platform

Many FORTRAN programs are written with the assumption that a very large memory is available. This is because most machines above the PC class have virtual operating systems. A standard "error-free" translation of a statement like the following:

```
DIMENSION A(10000,20),B(10000,20)
```

would be:

```
static float a[20][10000],b[20][10000];
```

The fact that this is syntactically error-free is of little significance since no C compiler presently available for the PC will accept it. Faced with this problem, what do you do? In the brute force approach, you would probably go through the code — maybe all 10000 lines of it — and change the references to arrays A and B to some sort of disk reference. You would probably also analyze how these arrays are referenced so that you could make the disk accesses as efficient as possible.

In processing large programs on the limited memory model of the PC-DOS platform, PROMULA FORTRAN takes a virtual memory approach. In PROMULA FORTRAN, you can tell the translator that all variables larger than a certain number of bytes should be treated as virtual memory disk variables. The runtime library has a very efficient set of virtual memory management routines, and the translator replaces references to the specified variables with function calls to the virtual memory functions. The subscript calculations are replaced by virtual memory address calculations.

Thus, PROMULA FORTRAN allows you to bring an entire class of FORTRAN programs to the PC which cannot now be processed easily by any other product available.

1.8 Dealing with FORTRAN Dialect Problems

Other translators deal primarily with FORTRAN 77, require that tokens contain no blanks and be separated, and treat FORTRAN statement names as reserved words. FORTRAN codes not meeting these specifications must be changed by the user manually. Our experience has been that many FORTRAN programs are written in FORTRAN 66 or in some mixed dialect of FORTRAN 66 and FORTRAN 77. Since FORTRAN has no keywords and explicitly ignores all blanks in its source statements, many perfectly readable FORTRAN programs do not meet the above requirements.

PROMULA FORTRAN translates FORTRAN 66 as well as FORTRAN 77 programs. The user can control those aspects of the two languages which conflict by selecting options in the translation process.

The translator accepts split tokens, token sequences without separators, blank lines, comment lines within continuation sequences, and other potential translation ambiguities. You do not have to "clean up the code" to use PROMULA

FORTRAN. So, if you have an old 66 program punched on a 026 keypunch with a drum card on as few cards as possible and you can get it onto a disk, PROMULA FORTRAN can process it for you.

Also, PROMULA FORTRAN knows the difference between

```
DO10I=1,5 and DO10I=1.5
```

or

```
DO 10 I = 1,5 and DO 10 I = 1.5
```

and translates them all correctly.

PROMULA FORTRAN deals with all of the "violations" of standard character manipulation including hiding character values in arithmetic variables, mixing character and noncharacter variables in COMMON blocks, equivalencing character variables with arithmetic variables, and assuming that CHARACTER*N and CHARACTER*1(N) have the same memory representations.

Finally, in cases where different versions of FORTRAN have conflicting features or conventions a dialect selection option switch can be used to select the desired set. The particular dialects which the compiler supports are as follows:

1.9 Dealing with C Types and FORTRAN Types

PROMULA FORTRAN gives you access to all of the following standard C types:

signed char
unsigned char
short signed int
short unsigned int
long signed int
long unsigned int
float
double

and to float complex and double complex structured types.

Though PROMULA FORTRAN has the usual set of default FORTRAN types, you can specify which FORTRAN type should connect to which C type. If your dialect of FORTRAN includes nonstandard types such as BOOLEAN or BYTE or INTEGER*3, then you can include these. In addition, FORTRAN compilers differ as to whether logical types are bit sequences or simply TRUE/FALSE values. You may also select this. If bit sequences are assumed, the "logical" operators become bit-manipulation operators.

1.10 Dealing with FORTRAN Input/Output in C

The approach taken to FORTRAN input/output statements by PROMULA FORTRAN is straightforward: the C programs should behave identically to their FORTRAN counterparts. All standard I/O statements are accepted including long FORMAT statements. These tend to be rejected by C compilers because their initialization strings exceed the compiler limit on the length of individual static strings. C input/output is managed via calls to functions in the C library. C has no input/output statements as such. FORTRAN input/output statements are translated into calls to functions in the PROMULA FORTRAN runtime library. These functions perform input/output the way FORTRAN does.

Though the general form of input/output statements is quite standard throughout the FORTRAN dialect community, no two FORTRAN compilers have the identical set of options. This is the major area in which dialects differ. With the PROMULA FORTRAN dialect manager you may describe the individual options associated with your particular dialect. In addition,

you may specify the form of the function to be referenced in the translation. Then you can simply write the needed function and include it in your version of the runtime library.

1.11 Runtime Library

The PROMULA FORTRAN runtime library is a set of approximately 250 functions designed for use with the PROMULA FORTRAN translator. It may also be used by those FORTRAN programmers who wish to program in C, but who do not wish to give up the input/output conventions, formatting controls, and intrinsic functions which they have grown used to.

Initially, C codes using this library can be produced by translating FORTRAN programs into C using the PROMULA FORTRAN translator. Once in C, the programs may then be maintained by using these functions.

If there is one certainty, it is that no two FORTRANs behave in the same way, especially with regard to their runtime libraries. Thus, if your conventions differ from the ones used here, you may alter the library code. Alternatively, your version of FORTRAN may contain statements which require runtime support not included in this library. In this case, you can add the additional functions needed.

1.12 Dealing with Common Blocks

If there is any aspect of FORTRAN that can destroy the validity of a translation, it is COMMON blocks — especially when combined with EQUIVALENCE statements. Every storage trick ever conceived gets used in the nuances of changing COMMON block definitions through a large FORTRAN program. There is no best way to translate COMMON blocks. Some compilers, for example, insert extra bytes to achieve various types of alignments, and programmers using these compilers will take that fact into account without any warning or comment for the user.

PROMULA FORTRAN translates COMMON blocks in one of four ways. It is up to you to select the appropriate one depending upon your needs and preferences:

1. The default is to declare the COMMON block identifiers simply as external void pointers and then to assign them locally to a structure pointer whose members are defined in the same manner as the COMMON definition in the routine. This technique works in all cases except where alignments are needed.

2. In cases where COMMON blocks are always defined in the same manner, the common blocks are simply defined as external structures. This gives more efficient code than the technique above.

3. The most efficient technique can be used when no games are played with the common blocks at all. Then, the blocks themselves are removed and the variables within them become external variables directly.

4. The final technique is used when an exact block layout is required. The COMMON block is declared as an external pointer to a char. The variable positions within the block are calculated with user-supplied alignment and size specifications. Actual variable references then become references to the COMMON block name plus the calculated position.

1.13 Allocation of Local Variables

Most FORTRAN compilers allocate a fixed unique storage location to all variables, be they COMMON or local. The equivalent allocation in C is called "static", and by default all variables in the translation are declared as static variables. C, however, has two additional storage allocation methods — auto and dynamic.

Auto variables are stored on the program stack automatically when a function is called and are automatically removed when the function is exited. The advantage of these variables is that they occupy memory only when needed and their physical allocation is quick and transparent to the C program. The disadvantage of auto variables is that the program stack tends to be quite short; thus, the number and size of auto variables are very limited.

Dynamic variables are stored on the heap via explicit calls to a function and must also be freed explicitly via a function call. The disadvantage of dynamic variables is that their allocation is slow as compared to auto variables, and the allocation itself must be done explicitly. The advantage is that the heap is generally quite large.

Using PROMULA FORTRAN you have complete control over the allocation of variables in the translation.

1.14 A Sample Translation to C

The output from PROMULA FORTRAN does not look like machine translation. To present some of the design features of the translator, consider the following example program which computes the mean and variance of a set of values.

Input	Output	Notes

`SUBROUTINE EX001(VAL,N,XBAR,VAR)`
`  DIMENSION VAL(N)`
`  XBAR=0.0`
`  VAR=0.0`
`  DO 10 J = 1,N`
`  XBAR = XBAR + VAL(J)`
`10 CONTINUE`
`  XBAR = XBAR/N`
`  DO 15 J = 1,N`
`  S = VAL(J) - XBAR`
`  VAR = VAR + S*S`
`15 CONTINUE`
`  VAR = VAR/(N-1)`
`  RETURN`
`  END`
	`void ex001(float *val,int n,float *xbar,float *var)`
	`{`
	`auto int j;`
	`auto float s;`
	`  *xbar = *var = 0.0;`
	`  for(j=0; j<n; j++) *xbar += *(val+j);`
	`  *xbar /= n;`
	`  for(j=0; j<n; j++) {`
	`  s = *(val+j)-*xbar;`
	`  *var += (s*s);`
	`  }`
	`  *var /= (n-1);`
	`}`
		(1)
		
		(2)
		(3)-(6)
		(5)
		(3) (4)
		
		(5)
		(7)
		(5)

Note (1) that the parameter n is not declared as a pointer, since it is not changed within the routine. PROMULA FORTRAN uses what are called "prototypes" of subprogram arguments so that it can generate optimal calling sequences. These "prototypes" may be specified by the user or may be determined internally by the translator. The above was internally determined by the translator.

Note (2) that C allows multiple assignments to the same value to be written together. The translator looks for such assignments and combines them whenever possible.

Note (3) that in FORTRAN the default base for a subscript is 1. Thus, all DO loops which generate subscripts tend to start at 1. In C, however, subscripts start at zero. This fact makes for much more efficient code. The translator looks for DO loops whose only purpose is to move through array subscripts and reduces their range to start at zero, thus producing a very natural looking for-statement and optimizing subscript expressions.

Note (4) that C has "++" and "--" operators which take advantage of the fact that most computers have increment and decrement operators. The translator uses these operators whenever possible.

Note (5) that C has operators like "+=", "-=", "*=", "/=", etc. The use of these operators ensures that the address of the left-hand-side of the assignment will only be computed as often as necessary. PROMULA FORTRAN uses these operators.

Note (6) that the DO loop running to statement 10 in the FORTRAN code is collapsed into a single compound statement, and that the now unneeded statement label is removed.

Note (7) that though the DO loop statements in loop 15 cannot be reduced to a single statement, the statement label can still be removed.

In summary, PROMULA FORTRAN looks for every opportunity to simplify and optimize the translation and to make it look as natural as possible in the C language form.

2. COMMAND LINE

To use PROMULA.FORTRAN in its simplest mode, type its name, pfc, followed by the name of the FORTRAN file that you wish to process and then press the [Enter] or [Return] key. This operation assumes that the following files are stored either in the default path, or more typically, in the path specified via the operating system:

pfc[.exe] The PROMULA.FORTRAN executable

pfc.pak The description of the FORTRAN and C dialects to be used.

To compile codes you need to use whatever conventions are appropriate for the C compiler that you are using. Typically this means that the file

```
fortran.h
```

should be copied into the same directory or disk which contains the other include files for the compiler, and the appropriate PROMULA.FORTRAN runtime library file

`pfcmsc.lib` or `libpfc.a` or `pfcvms.olb`

should be copied into the same directory which contains the other library files for the linker.

Note that details on the installation and placement of the PROMULA.FORTRAN files are included with the installation instructions which accompanied the PROMULA.FORTRAN product.

There are three ways in which to modify the behavior of PROMULA.FORTRAN. The easiest way is via command line switches. This chapter discusses the use of PROMULA.FORTRAN via its command line switches.

In addition, you may supply a configuration file which gives you detailed control over particular aspects of the translation process and specifies how individual parameters associated with particular subroutines and functions are to be processed. The use of configuration files is discussed in Chapter 3.

Finally, you may replace the entire dialect description file

pfc.pak

that is controlling the PROMULA.FORTRAN translations. Special dialect definition files may be obtained from Great Migrations LLC upon request.

2.1 Command Line Syntax

The syntax for using PROMULA.FORTRAN from the operating system level is as follows:

```
PFC filename opt1 opt2 opt3 ....
```

Where:

`filename` is the name of the FORTRAN file to be converted, optionally preceded by a drive and path specification. If no extension is supplied, an extension of `.for` is assumed.

`opti` is a single character or group of characters (in upper or lower case) indicating a particular switch followed by any particular information to be associated with that switch. The switch letter(s) plus its associated information may contain no embedded blanks. The various options are separated by blanks.

The following is an alphabetic listing of the command line switches, the number of the section in which each is discussed, and a brief description. The remaining sections of this chapter will discuss these switches in detail.

Switch	Options	Section	Characteristic effected by switch
B	c,f,o	2.2	Specify your C output
C	l,s	2.3	Treatment of short arithmetic
C	0,1,2	2.3	Casting level
CF	num	2.4	Details of C output format
CH	d,r,s,v	2.5	Treatment of character variables
CM	0,1,2	2.6	Appearance of comments in C output
D	c,r	2.7	Treatment of data initializations
DB		2.8	Debugging information in C output
EL	0,1,2,3	2.9	Error message level
EP		2.9	Echo pseudo-code produced
ER	0,1,2,3,4	2.10	Treatment of syntax errors
ES		2.9	Annotated listing of source code
ET		2.9	Annotated listing of C output
EX		2.9	Echo symbol references by line number
EZ		2.9	Intermediate symbols table
F	snumb,t,f,v,9	2.11	Input format used
FI	s,l	2.12	FORTRAN INTEGER type
G	name	2.13	Specify a globals file
G	v,s,p,r,d,a	2.14	COMMON variable convention
G	pc,ps,pl,pd	2.14	Overall alignment control
I	name	2.15	Inline functions file
I	s,l	2.16	Target C int type
K	a,s	2.17	Treatment of internally generated constants
L	num	2.18	Maximum output line width
L	m,s	2.19	Link time processing of COMMON initializations
L	n,o	2.20	Inclusion of line numbers for debugging
M	dialect	2.21	Source language dialect
N	*,num	2.22	Nesting indentation to be used in output
NC	num	2.23	Inline comments C output margin width
NU	0,1,2	2.24	Upper braces placement convention in C output
NL	1,2	2.24	Lower braces placement convention in C output
O	name	2.25	Name of file to receive the C output
O	m,s	2.26	Splitting of C output into separate files
P	numb	2.27	Miscellaneous prototyping controls
P+	numb	2.27	Additional prototyping control
PA	name	2.28	Listing append filename
PH	numb	2.28	Listing file page height
PN	name	2.28	New listing filename
PW	numb	2.28	Listing file page width
QI	numb	2.29	Size of compacted statement storage
QE	numb	2.29	Size of the line number table
QD	numb	2.29	Size of a data block
QX	numb	2.29	Size of external information storage
QH	numb	2.29	Size of include file information storage
QW	numb	2.29	Word size of source platform
R	name	2.30	Specify a configuration file
SA	num	2.31	Auto storage threshold

Switch	Options	Section	Characteristic effected by switch
SD	num	2.31	Dynamic storage threshold
SS	num	2.31	Static storage threshold
SV	num	2.31	Virtual storage threshold
SZ	num	2.31	Dynamic virtual threshold
T	0,1,2	2.32	FORTRAN dialect DO loop assumption
T	a,s	2.33	Treatment of internally generated temporaries
UP	num	2.34	PUNCH statement unit number
UR	num	2.34	READ or ACCEPT statement unit number
UW	num	2.34	WRITE or PRINT statement unit number
W	name	2.35	Name of file to receive prototype definitions
Y	1,2	2.36	Miscellaneous control flags
X,Y	a	2.37	Treatment of multiple assignments convention
X,Y	b	2.38	Treatment of single statement nesting brace
X,Y	c	2.39	Constants reduction optimization
X,Y	ch	2.40	Character optimization
X,Y	d	2.41	Treatment of FORTRAN "D" debugging statements
X,Y	f	2.42	Use of printf-style formatting
X,Y	i	2.43	Initialization check for auto variables
X,Y	l	2.44	DO loop counter reduction optimization
X,Y	p	2.45	Subprogram argument type checking
X,Y	r	2.46	Single precision real arithmetic
X,Y	s,sv,sf	2.47	Subscript pointer notation
X,Y	u	2.48	Unformatted write optimization
X,Y	v	2.49	Subprogram call-by-value arguments
X,Y	$	2.50	Dollar signs as initial symbols in identifiers
Z	name	2.51	Location of FORTRAN files to be included

2.2 Specifying your C output bias — Bc, Bf, Bo

In the October, 1988 issue of *Computer Language*, Mark Davidson reviews PROMULA FORTRAN and declares it the clear winner over its competitors; however, he says "Although the code produced by the translator is not immediately understandable ... ". Since that review, we have surveyed the users of PROMULA FORTRAN extensively to determine how they feel the output code can be made as understandable as possible and have found no clear consensus. However, users can be divided into three broad categories:

1. Those who want to continue using their present FORTRAN dialect as their programming language. For those users the C output is of no importance as such. It is merely an intermediate step. It should be designed to compile as quickly as possible and to produce an efficient as possible executable.

2. Those who are presently FORTRAN programmers, but who want to (or must) become C programmers. For them the C output should be as close to the original FORTRAN as possible to ease the transition.

3. Those who are C programmers who must now take over a FORTRAN code. For them the C output should look as much like a standard C program as possible.

This general issue is referred to as "user bias". Those users who want optimized code have the "optimized" bias, those who want FORTRAN-like code have the "FORTRAN" bias, and those who want C-like code have the "C" bias. When using PROMULA FORTRAN this output bias can be selected via a single command line switch. The default bias is the FORTRAN bias which can be selected via the Bf switch. The C and optimized biases are, however, fully supported by PROMULA FORTRAN and may be activated by the Bc and Bo command line switches.

When one of the bias switches is used, it should always go first on the command line, since it controls the selection of various other switch values. Mixed bias can be produced by following the bias switch itself with other switches.

Regardless of the user bias, the output of PROMULA FORTRAN does not look like machine translation. To present some of the design features of the different biases, consider the following example program which computes the mean and variance of a set of values. The figure below shows the original FORTRAN source code along with the default translation produced by PROMULA FORTRAN for the FORTRAN and C biases.

FORTRAN SOURCE	FORTRAN BIAS TRANSLATION	NOTES
`SUBROUTINE EX001(VAL,N,XBAR,VAR)`	`void ex001(val,n,xbar,var)`	1,2,3
`DIMENSION VAL(N)`	`long n;`	4
	`float val[],*xbar,*var;`	
	`{`	
	`static long j;`	2,3,5
	`static float s;`	3,5
	`WRITE(OUTPUT,VFMT,"(1x,a,//)",CSTR,"ANALYSIS",0);`	6
	`*xbar = 0.0;`	7
`WRITE(*,'(1x,a,//)') "ANALYSIS"`	`*var = 0.0;`	
`XBAR = 0.0`	`for(j=0; j<n; j++) {`	4,8-11,13
`VAR = 0.0`	`*xbar = *xbar+val[j];`	
`DO 10 J = 1,N`	`}`	
`XBAR = XBAR + VAL(J)`	`*xbar = *xbar/n;`	12,13
`10 CONTINUE`	`WRITE(OUTPUT,VFMT,"(6H Mean=,F10.3)",REAL4,*xbar,0);`	11
`XBAR = XBAR / N`	`for(j=0; j<n; j++) {`	6
`WRITE(*,'(6H Mean=,F10.3)')XBAR`	`s = val[j]-*xbar;`	8,9,13
`DO 15 J = 1,N`	`*var = *var+s*s;`	4
`S = VAL(J) - XBAR`	`}`	11
`VAR = VAR + S * S`	`if(*var != 0.0) *var = *var/(n-1);`	12,13
`15 CONTINUE`	`WRITE(OUTPUT,LISTIO,CSTR,"Variance = ",REAL4,*var,0);`	10,11
`IF(VAR.NE.0.0) VAR = VAR/(N-1)`	`return;`	6
`WRITE(*,*) "Variance = ",VAR`	`}`	
`RETURN`		
`END`		

FORTRAN SOURCE	C BIAS TRANSLATION	NOTES
`SUBROUTINE EX001(VAL,N,XBAR,VAR)`	`void ex001(val,n,xbar,var)`	1,2,3
`DIMENSION VAL(N)`	`long n;`	4
	`float val*,*xbar,*var;`	
	`{`	
	`static long j;`	2,3,5
	`static float s;`	3,5
	`printf(" ANALYSIS\n\n\n");`	6
	`*xbar = *var = 0.0;`	7
`WRITE(*,'(1x,a,//)') "ANALYSIS"`		
`XBAR = 0.0`	`for(j=0; j<n; j++) *xbar += *(val+j);`	4,8-11,13
`VAR = 0.0`		
`DO 10 J = 1,N`		12,13
`XBAR = XBAR + VAL(J)`		11
`10 CONTINUE`	`*xbar /= n;`	6
`XBAR = XBAR / N`	`printf(" Mean=%10.3f\n",*xbar);`	8,9,13
`WRITE(*,'(6H Mean=,F10.3)')XBAR`	`for(j=0; j<n; j++) {`	4
`DO 15 J = 1,N`	`s = *(val+j)-*xbar;`	11
`S = VAL(J) - XBAR`	`*var += (s*s);`	12,13
`VAR = VAR + S * S`	`}`	10,11
`15 CONTINUE`	`if(*var != 0.0) *var /= (n-1);`	6
`IF(VAR.NE.0.0) VAR = VAR/(N-1)`	`printf("Variance = %16.6E\n",*var);`	
`WRITE(*,*) "Variance = ",VAR`	`return;`	
`RETURN`	`}`	
`END`		

In general, the objective of the FORTRAN bias is to make the C output correspond as closely as possible to the FORTRAN original, while still producing correct C. Extensive use is made of defined symbols to achieve this goal. In addition, every effort is taken to keep the source and output statements in correspondence. The objectives of the C bias, on the other hand, are to make the output as C-like as possible. No attempt is made to maintain a statement by statement correspondence and, whenever possible, FORTRAN I/O statements, along with associated FORMAT specifications, are translated into traditional C I/O statements.

The lines in the figure above are annotated with note numbers. The notes are as follows:

Note (1) that the parameter n is not declared as a pointer, since it is not changed within the routine. PROMULA FORTRAN uses what are called "prototypes" of subprogram arguments so that it can generate optimal calling sequences. These prototypes may be specified by the user or may be determined internally by the translator. The above was internally determined. Technically, all parameters in FORTRAN are passed as pointers. For FORTRAN codes that make use of this

fact and mix types across calls, PROMULA FORTRAN may be told to make all parameters pointers. Alternatively, the user can specify individual prototypes via a separate file.

Note (2) that ANSI FORTRAN requires that default integers and reals both occupy the same amount of memory, typically 4 bytes. Therefore, by default PROMULA FORTRAN declares FORTRAN integers as long. If on your platform C ints are already long or if in your source FORTRAN dialect FORTRAN integers are short, PROMULA FORTRAN may be told to translate INTEGER as int or as short.

Note (3) that in C all symbols must be defined prior to their use; therefore, PROMULA FORTRAN declares all parameters and local variables explicitly. Even in the FORTRAN bias, no attempt is made to maintain a direct correspondence between FORTRAN declarations and C declarations.

Note (4) that the declaration of the `val` vector differs between the C and the FORTRAN biases. In the FORTRAN bias the C brackets notation is used which, though different from standard FORTRAN notation, is preferred by many to the traditional C pointer notation used by the C bias. This notation may be controlled via command line switches.

Note (5) that most implementations of FORTRAN treat variables as though they were static — i.e., each variable is assigned its own memory location which remains undisturbed even when the routine containing it is not active. The user may specify that variables be declared as "auto". In general, the user of PROMULA FORTRAN has extensive control over the allocation and memory status of variables.

Note (6) that C and FORTRAN have very different ways of specifying coded write conversions. For the FORTRAN bias, all of the original FORTRAN machinery is maintained. The actual WRITE statement is translated into a C WRITE statement which consists of a series of keywords followed by the parameters associated with those keywords. In the C bias, FORTRAN WRITE statements are translated into the C printf or fprintf functions whenever possible. When not possible, the C bias uses the same translation as the FORTRAN bias. Note in the above that both list-directed and FORMAT-controlled statements are processed. For formatted statements the equivalent printf specification is used. For list directed conversions PROMULA FORTRAN uses a default set of specifications which may be modified by the user.

Note (7) that C allows multiple assignments to the same value to be written together. Under the C bias, the translator looks for such assignments and combines them whenever possible. The FORTRAN bias does not by default perform these combinations, since they destroy the correspondence between source and output statements. This feature is also controllable via a command line switch.

Note (8) that in FORTRAN the default base for a subscript is 1. Thus, all DO loops which generate subscripts tend to start at 1. In C, however, subscripts start at zero. This fact makes for much more efficient code. The translator looks for DO loops whose only purpose is to move through array subscripts and reduces their range to start at zero, thus producing a very natural-looking for statement and optimizing subscript expressions. The DO loop reduction feature is on for all biases by default, but it may be turned off via a command line switch.

Note (9) that C has "++" and "– –" operators which take advantage of the fact that most computers have increment and decrement operators. The translator uses these operators whenever possible for all biases.

Note (10) that C allows any conditional statement to form a compound statement with a single statement, while FORTRAN allows this only for the IF statement. In the C bias this compounding is performed whenever possible; while in the FORTRAN bias a compound statement is formed only if the source was compound.

Note (11) that C has operators like "+=", "-=", "*=", "/=", etc. The use of these operators ensures that the address of the left-hand side of the assignment will not be computed more often than is necessary. The C bias uses the operators, the FORTRAN bias does not.

Note (12) that the statement labels and CONTINUE statements ending the two DO loops are not needed for any other purpose. Unneeded statement labels are always removed.

2.3 Arithmetic Conversions — CL, Cs, C0, C1, C2, C3

In C, if a binary operator like + or * has operands of different types the lower type is promoted to the higher type automatically by the compiler. Alternatively, if operands of the same type are combined then the result is of that type. As PROMULA FORTRAN processes source code, it keeps track of the types of all operands involved in expressions and determines when any conversions need to be performed. Normally, however, it only shows these conversions in the C output if they would not be performed automatically by the C compiler.

Alternatively, the convention that operands of the same type are not promoted can cause trouble for short integer arithmetic, since most FORTRAN compilers promote all short integer calculations to long.

Explicit conversions in C can be forced in any expression with a cast which looks as follows:

(type) expression

This cast converts the expression to the indicated type. The C casting level option allows one to control which casts are "forced" in the C. Both aspects of promotion are controlled via this switch. The switch itself may occur more than once on the command line. Its individual settings are as follows:

Setting	Meaning
C0	Specifies that all promotions between operands of different types be forced in the C output.
C1	Specifies that all promotions between different operands involving any integer types be forced, but that conversions between float and double not be forced.
C2	Specifies that only those conversions between fixed point and floating point be forced, but that other conversions not be forced.
C3	Is the default and specifies that only those casts needed to maintain the integrity of a calculation be maintained.
Cs	Specifies that short integer calculations are to be done using short arithmetic.
CL	Is the default and specifies that short integer calculations are to be done using long arithmetic.

As an example, consider the following simple FORTRAN code that computes the square of a weighted mean.

```
      SUBROUTINE DEMO(VAL,NVAL,WEIGHT,SQ)
      DIMENSION VAL(*)
      INTEGER*2 NVAL,WEIGHT,POW
      XBAR = 0
      POW = 2
      DO 10 I = 1,NVAL
      XBAR = XBAR + VAL(I)
   10 CONTINUE
      XBAR = XBAR/(NVAL*WEIGHT)
      SQ = XBAR ** POW
      RETURN
      END
```

The default C output for this example looks as follows:

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
```

```
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= ((long)nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

Notice that there are only two casts shown. The long cast on the product between `nval` and `weight` is needed in case this product exceeds the maximum value of a short integer. This topic is discussed below in a subsection on arithmetic with short integer variables. The other cast is on the variable `Pow` in the exponentiation. Though exponentiation is a binary operator in FORTRAN, it is not in C; therefore, in this cast the double is necessary in order to make the parameter for the `pow` function have the proper type. The `xbar` parameter does not require a forced cast because C automatically promotes a float to a double when it passes it by value.

The same output using the Cs switch is as follows:

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= (nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

It is identical with the above except that the cast on the short integer conversion is not shown. Be careful, this version might produce an incorrect result. See the subsection below for a detailed discussion of arithmetic with short integer variables.

Let us now go to extremes. The following is the same output using the C0 cast which requests that all casts be explicitly shown.

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L; i<=(long)nval; i++) xbar = (float)((double)xbar+
      (double)*(val+(short)i-1));
    xbar = (float)((double)xbar/(double)((long)nval*(long)
      weight));
    *sq = (float)pow((double)xbar,(double)Pow);
}
```

In all likelihood you would never want to run PROMULA FORTRAN in this mode unless you are interested in seeing all of the "promoting" that actually goes on. Notice for example that all floating point calculations are promoted to double and then reduced back to float. Note that constants also get promoted; thus, in the `for` statement the value of 1 assigned to `i` is now shown as `1L`. Finally, note that the long cast forced on `nval` by the CL setting in fact causes a long cast on `weight`. This is how this convention forces short arithmetic to be done as long arithmetic. It might be instructive to see this example again, not with C0 but with Cs which will not force these long promotions.

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L; i<=(long)nval; i++) xbar = (float)((double)xbar+
      (double)*(val+(short)i-1));
    xbar = (float)((double)xbar/(double)(nval*weight));
    *sq = (float)pow((double)xbar,(double)Pow);
}
```

The other effect of forcing the casts is that this process blocks the DO loop reduction algorithm. Thus, in the initial version, the loop was reduced to start at zero; but now it starts at one. We decided to do this since forcing the casting level probably means that you are very concerned about the arithmetic being performed. Since loop reduction performs additional integer arithmetic, we turn it off. As one might fear, the cast to double is too late. The division is done at the integer*2 level and the possibly overflowed result is promoted to double. Thus, using C0 to fix the short arithmetic problem only shows us what the problem is.

Moving up to C1, the following example shows the C output using the C1 casting level.

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L; i<=(long)nval; i++) xbar += *(val+(short)i-1);
    xbar /= (double)((long)nval*(long)weight);
    *sq = pow(xbar,(double)Pow);
}
```

As specified, the float-double casts are no longer forced, but all fixed point casts are forced. At the C2 casting level, the following is the result.

```
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= (double)((long)nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

Now only the "mixed-mode" cast is forced. Integer arithmetic is again free, so the loop reduction is also allowed.

2.3.1 Arithmetic with Short Integer Variables

In moving from one environment to another, one must always be concerned with the accuracy of floating point arithmetic; however, there is also a real problem with fixed point arithmetic even when dealing with identical word sizes. This section concerns itself with short integer arithmetic in non-short integer environments. There is a real semantics issue here: the same program compiled in different environments behaves differently even though these environments have the same word size. Consider the following FORTRAN program which does short integer additions, multiplications, and divisions in a variety of contexts.

```
      PROGRAM VARI2
      INTEGER*2 I1,I2,I3,I4
      INTEGER*2 ISUM,IPROD,IQUOT
      INTEGER*4 JSUM,JPROD,JQUOT
      REAL*4 RSUM,RPROD,RQUOT
      REAL*8 DSUM,DPROD,DQUOT
      I1 = 20000
      I2 = 30000
      I3 = 200
      I4 = 300
      ISUM = I1 + I2         <-- Note addition overflow
      IPROD = I3 * I4        <-- Note multiplication overflow
      IQUOT = (I1 + I2) / I4 <-- Note intermediate overflow
      JSUM = I1 + I2
      JPROD = I3 * I4
      JQUOT = (I1 + I2) / I4
      RSUM = I1 + I2
      RPROD = I3 * I4
      RQUOT = (I1 + I2) / I4
      DSUM = I1 + I2
      DPROD = I3 * I4
      DQUOT = (I1 + I2) / I4
      WRITE(*,'(24H Short Integer Results: ,3I13)')
     +   ISUM,IPROD,IQUOT
      WRITE(*,'(24H Long Integer Results:  ,3I13)')
     +   JSUM,JPROD,JQUOT
      WRITE(*,'(24H Short Real Results:    ,3F13.5)')
     +   RSUM,RPROD,RQUOT
      WRITE(*,'(24H Long Real Results:     ,3F13.5)')
     +   DSUM,DPROD,DQUOT
      STOP
      END
```

In running this example with various FORTRAN compilers, always on machines with 16-bit short integers (VAX, IBM mainframe, IBM PC), we have obtained the following three results:

1. Universal promotion to long

```
Short Integer Results:        -15536        -5536          166
Long Integer Results:          50000        60000          166
Short Real Results:      50000.00000  60000.00000    166.00000
Long Real Results:       50000.00000  60000.00000    166.00000
```

2. No automatic promotion to long

```
Short Integer Results:        -15536        -5536          -51
Long Integer Results:         -15536        60000          -51
Short Real Results:     -15536.00000  -5536.00000    -51.00000
Long Real Results:      -15536.00000  -5536.00000    -51.00000
```

3. Selective promotion to long

```
Short Integer Results:        -15536        -5536          166
Long Integer Results:          50000        60000          166
```

```
 Short Real Results:      50000.00000  -5536.00000    166.00000
 Long Real Results:       50000.00000  -5536.00000    166.00000
```

In reviewing these results, a negative number means that a short integer overflow has occurred. The typical FORTRAN result is the first one. In this instance, the output of all integer calculations is a long. That result is then converted to the desired result type. Notice that even the intermediate addition in the division example is calculated as a long.

In the second case, the result of any integer calculation is also always short, regardless of the surrounding context. This type of result is unusual for mainframe FORTRANs and is common for PC FORTRANs. Note that Microsoft FORTRAN allows the user to select which type of convention is to be followed as a side-effect of the "WORDSIZE" metacommand.

The third case is strange and difficult to deal with. The particular result above can be gotten from VS FORTRAN. Note that in an integer context, universal promotion to integer is followed. Also, in all cases the intermediate addition result is promoted to long. But for some reason the multiplication result is allowed to overflow while the addition result is not.

In designing PROMULA FORTRAN we allow the user to select whether he wants universal promotion to long or no automatic promotion to long. We have no provision for selective promotions. Note that selective and universal promotion differ only in overflow conditions, so users from such environments should use the universal promotion to long convention. The default convention is universal promotion. No automatic promotion is selected via the "CS" command line switch.

From a readability standpoint, the best convention is unfortunately not the cleanest one. In C, the results of all short binary operators are short. The only way to force C to produce a long result is to convert the arguments to long prior to the calculation. The C output for the assignments in the above FORTRAN program under the universal promotion convention is as follows:

```
void main(argc,argv)
int argc;
char* argv[];
{
static int i1,i2,i3,i4,isum,iprod,iquot;
static double dsum,dprod,dquot;
static long jsum,jprod,jquot;
static float rsum,rprod,rquot;
    ftnini(argc,argv);
    i1 = 20000;
    i2 = 30000;
    i3 = 200;
    i4 = 300;
    isum = (long)i1+i2;
    iprod = (long)i3*i4;
    iquot = ((long)i1+i2)/i4;
    jsum = (long)i1+i2;
    jprod = (long)i3*i4;
    jquot = ((long)i1+i2)/i4;
    rsum = (long)i1+i2;
    rprod = (long)i3*i4;
    rquot = ((long)i1+i2)/i4;
    dsum = (long)i1+i2;
    dprod = (long)i3*i4;
    dquot = ((long)i1+i2)/i4;
    ftnopen(6,FILEN,"EX002.OUT",9,STATUS,"NEW",0);
    fprintf(LUN(6)," Short Integer Results: %13d%13d%13d\n",isum,iprod,iquot);
    fprintf(LUN(6)," Long Integer Results:  %13ld%13ld%13ld\n",jsum,jprod,jquot);
    fprintf(LUN(6)," Short Real Results:    %13.5f%13.5f%13.5f\n",rsum,rprod,rquot);
    fprintf(LUN(6)," Long Real Results:     %13.5f%13.5f%13.5f\n",dsum,dprod,dquot);
    exit(0);
}
```

Note that in each case the left–hand argument is converted to long; thus, causing the entire expression to be evaluated in that manner. The result of running this version is shown below.

```
 Short Integer Results:        -15536        -5536          166
 Long Integer Results:          50000        60000          166
 Short Real Results:      50000.00000  60000.00000    166.00000
 Long Real Results:       50000.00000  60000.00000    166.00000
```

Note that it agrees with the universal promotion to long result shown above.

The simpler alternative is shown below. No conversions to long are made. Thus, overflows occur in every expression.

```
void main(argc,argv)
int argc;
char* argv[];
{
static int i1,i2,i3,i4,isum,iprod,iquot;
static double dsum,dprod,dquot;
static long jsum,jprod,jquot;
static float rsum,rprod,rquot;
    ftnini(argc,argv);
    i1 = 20000;
    i2 = 30000;
    i3 = 200;
    i4 = 300;
    isum = i1+i2;
    iprod = i3*i4;
    iquot = (i1+i2)/i4;
    jsum = i1+i2;
    jprod = i3*i4;
    jquot = (i1+i2)/i4;
    rsum = i1+i2;
    rprod = i3*i4;
    rquot = (i1+i2)/i4;
    dsum = i1+i2;
    dprod = i3*i4;
    dquot = (i1+i2)/i4;
    ftnopen(6,FILEN,"EX002.OUT",9,STATUS,"NEW",0);
    fprintf(LUN(6)," Short Integer Results: %13d%13d%13d\n",isum,iprod,iquot);
    fprintf(LUN(6)," Long Integer Results:  %13ld%13ld%13ld\n",jsum,jprod,jquot);
    fprintf(LUN(6)," Short Real Results:    %13.5f%13.5f%13.5f\n",rsum,rprod,rquot);
    fprintf(LUN(6)," Long Real Results:     %13.5f%13.5f%13.5f\n",dsum,dprod,dquot);
    exit(0);
}
```

The result of running this translation is shown below.

```
 Short Integer Results:        -15536        -5536          -51
 Long Integer Results:         -15536        -5536          -51
 Short Real Results:     -15536.00000  -5536.00000    -51.00000
 Long Real Results:      -15536.00000  -5536.00000    -51.00000
```

This result agrees with the no automatic promotion to long output from above, and is probably wrong for most applications. It should be noted that all other FORTRAN to C translators for the PC that we have reviewed produced only the no automatic promotion to long result.

If your program uses any short integer variables, be certain to consider this semantics problem.

2.4 Detailed C Output Format — CF1, CF2, CF4, CF8, CF16

Though the user has complete control over the actual content of the C output produced by PROMULA.FORTRAN by using various other command line switches and by modifying the content of the configuration file (see the chapter on the configuration file), there are still a few miscellaneous aspects of the look of the C output that can be controlled:

(1) the use of whitespace surrounding operators

(2) the amount of whitespace to be used in the switch statement

(4) the case of identifiers

(8) the amount of whitespace in structure and common definitions

(16) the display of constant parameters as constants

The values of the CF switches may also be combined to obtain composite effects; thus, any CF switch value between 0 and 31 is valid. The default setting is CF0.

Consider, for example, the following FORTRAN code fragment:

```
      SUBROUTINE TEST
      COMMON/ALPHA/IVAL,JVAL,KVAL
      PARAMETER(I3=3,I5=I3+2)
      IVAL = JVAL + KVAL - I5
      GOTO(10,20,30) IVAL
10    WRITE(*,*) 'IVAL = 1'
20    RETURN
30    STOP
      END
```

The default translation for this fragment is as follows:

```
void test()
{
#define i3 3
#define i5 (i3+2)
extern char Xalpha[];
typedef struct {
    int ival,jval,kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    Talpha->ival = Talpha->jval+Talpha->kval-i5;
    switch(Talpha->ival){case 1: goto S10;case 2: goto S20;case 3: goto S30;
      default: break;}
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

There is no whitespace within the expression "`Talpha->jval+Talpha->kval-i5`", the typedef and the switch statement are horizontally written to reduce whitespace, the identifiers in C have all been reduced to lower case, and the definition of `i5` is still shown as `i3+2`.

The same fragment with the CF1 switch is as follows:

```
void test()
{
#define i3 3
#define i5 (i3 + 2)
extern char Xalpha[];
typedef struct {
    int ival,jval,kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    Talpha->ival = Talpha->jval + Talpha->kval - i5;
    switch(Talpha->ival){case 1: goto S10;case 2: goto S20;case 3: goto S30;
      default: break;}
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

Now the expressions `Talpha->jval + Talpha->kval - i5` and `i3 + 2` have been widened, but the remainder of the C output is as before.

The same fragment with the CF2 switch is as follows:

```
void test()
{
#define i3 3
#define i5 (i3+2)
extern char Xalpha[];
typedef struct {
    int ival,jval,kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    Talpha->ival = Talpha->jval+Talpha->kval-i5;
    switch(Talpha->ival) {
        case 1: goto S10;
        case 2: goto S20;
        case 3: goto S30;
        default: break;
    }
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

In this version blanks are not inserted in expressions, but the switch statement is now vertical. Note that the appearance of the braces and indentation is controlled via the NU and NL switches, which are described in another section of this chapter. The CF4 switch produces the following result:

```
void TEST()
{
#define I3 3
#define I5 (I3+2)
extern char XALPHA[];
```

```
typedef struct {
    int IVAL,JVAL,KVAL;
} CALPHA;
auto CALPHA *TALPHA = (CALPHA*) XALPHA;
    TALPHA->IVAL = TALPHA->JVAL+TALPHA->KVAL-I5;
    switch(TALPHA->IVAL){case 1: goto S10;case 2: goto S20;case 3: goto S30;
      default: break;}
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef I3
#undef I5
}
```

Here all user supplied identifiers are shown in upper case (the traditional FORTRAN style) as opposed to lower case (the contemporary C style).

Using the CF8 switch gives the following C result:

```
void test()
{
#define i3 3
#define i5 (i3+2)
extern char Xalpha[];
typedef struct {
    int ival;
    int jval;
    int kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    Talpha->ival = Talpha->jval+Talpha->kval-i5;
    switch(Talpha->ival){case 1: goto S10;case 2: goto S20;case 3: goto S30;
      default: break;}
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

In this version the typedef statement used to define the structure of the COMMON block is shown vertically. Note again that the appearance of the braces and indentation is controlled via the NU and NL switches, which are described in another section of this chapter.

Finally, the CF16 switch produces the following result:

```
void test()
{
#define i3 3
#define i5 5
extern char Xalpha[];
typedef struct {
    int ival,jval,kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
```

```
    Talpha->ival = Talpha->jval+Talpha->kval-i5;
    switch(Talpha->ival){case 1: goto S10;case 2: goto S20;case 3: goto S30;
      default: break;}
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

In this version the definition of `i5` has been simplified to its simple value of `5`.

The effects of the switches may be combined by adding their values. For example, CF11 = 1 + 2 + 8 could be used to maximize whitespace. It produces the following result:

```
void test()
{
#define i3 3
#define i5 (i3 + 2)
extern char Xalpha[];
typedef struct {
    int ival;
    int jval;
    int kval;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    Talpha->ival = Talpha->jval + Talpha->kval - i5;
    switch(Talpha->ival) {
        case 1: goto S10;
        case 2: goto S20;
        case 3: goto S30;
        default: break;
    }
S10:
    WRITE(OUTPUT,LISTIO,STRG,"IVAL = 1",8,0);
S20:
    return;
S30:
    STOP(NULL);
#undef i3
#undef i5
}
```

With this setting the expressions are widened with whitespace and the switch and typedef statements have a vertical form; however, parameter values are still shown in their original form and identifiers are in lower case.

2.5 Treatment of CHARACTER Variables — CHd, CHr, CHs, CHv

By far the most difficult task to be faced by the processor is that of dealing with FORTRAN character manipulation. The problem is that, despite the efforts of the various standards committees, FORTRAN as a living computer language has no unified approach to character manipulation.

FORTRAN was originally designed to do "formula translation". It had little use for characters other than to label the results of calculations. FORTRAN 66 has no CHARACTER data type and has no character manipulation statements other than formatted I/O operations. Character data is simply hidden in whatever variable is convenient. Once a programmer stores character data in a "numeric" variable it is up to him to ensure that he does not use that variable for anything else.

As the language matured, a CHARACTER type was added and some minimal machinery was included to manipulate these types of variables. However, since there were many FORTRAN 66 programs still in existence, the use of the new CHARACTER type did not preclude using the old character management techniques as well. Things such as equivalencing numeric variables and character variables were completely valid.

When the 77 FORTRAN standards were established, the committee rightly concluded that the way in which languages like IBM FORTRAN level H had implemented characters was clearly not machine transportable and was not acceptable. Therefore, they made all FORTRAN 66 techniques of hiding character data in numeric variables illegal. In addition, they placed several restrictions on the placement and handling of character variables, all intended to improve the portability of the language.

After the FORTRAN 77 standard came out, however, no compiler vendor felt that he could afford to lose his customers by telling them that they had to convert all of their FORTRAN programs into the new standard; therefore, all compiler vendors put features into their compilers which allowed existing programs to still operate. The standards committee, however, had totally ignored the problem of what to do with existing programs. An aside here is that we first attempted to write a processor which would take older dialects of FORTRAN to the 77 standard. This effort proved impossible. The languages are simply incompatible. FORTRAN 77 does not have the openness of C.

Since there was no standard mixed 66 - 77 FORTRAN, all the vendors created their own. Still today, as new compilers are created, even for the PC, they differ in how they deal with this problem of 66 versus 77 FORTRAN. The problem is now compounded in that particular programs are being written not for either standard but rather for the incompatible hybrid combinations.

In designing PROMULA.FORTRAN we decided that the major barrier to using existing FORTRAN programs in a new machine environment was this incompatibility. If a program were written in pure FORTRAN 77, then any one of the existing compilers could probably process it with no problem. Unfortunately, few programs are written in pure 77 FORTRAN, and as time passes the possible hosts for these programs disappear.

Our treatment of CHARACTER data attempts to deal with all of the dialects with which we are familiar. If a program hides its characters in numeric variables, no problem. If a program mixes character and non-character variables in COMMON blocks or across subroutine parameters, no problem. If a program assumes that a CHARACTER*20 in one place is a CHARACTER*1(20) in another place, no problem. If it worked in the original, then it will work in the C version.

A character variable is treated simply as a sequence of chars. It has no other structure. A CHARACTER*4 is identical to a CHARACTER*2(2) or a CHARACTER*1(4) or a sequence of characters hidden in an INTEGER*4. The elements of a CHARACTER array are stored one after another with no intervening NULL characters or character counts or intermediate character pointers. This storage of character data is simple and allows the easy simulation of the various "tricks" allowed by the various dialects.

If FORTRAN had no subprograms there would be no problems. Unfortunately, FORTRAN subprograms allow for variable length characters, and the various system operations allow for the total mixture of character strings of any length. To deal with a variable character string, the length of that string must be known, since the storage technique for character variables itself does not have this information. This is an especially difficult problem, since the length of a given character string is defined not globally, but rather by the context of its use.

The solution is difficult to accept, but appears to be dictated by the needed generality of the translation. If a given program does not use variable length characters (CHARACTER*(*)) in subprograms, then there is no problem. Programs using FORTRAN 66 conventions and programs using FORTRAN 77 conventions can both be handled straightforwardly by translating character variables into C arrays of char. If a program does use the CHARACTER*(*) construct, however, then all references to character variables in the FORTRAN program are translated into a pair of values — a pointer to the start of the characters being manipulated and an integer value which defines the length of the character string at that point in the code. To distinguish these two cases the user must tell PROMULA.FORTRAN what assumptions his program is making about the availability of character string length information. In addition, the user must specify whether he is mixing character and non-character variables across subprograms.

To make life more difficult, there are three alternative ways in which the character address and character length can be joined:

(1) The address and length can become an ordered pair when the character information is passed to a subprogram;

(2) The character lengths can be saved and passed together at the end of each call;

(3) A separate entity, termed a "descriptor", can be formed which contains the address and length.

The second approach above is the default one taken by PROMULA.FORTRAN. It is the most common approach used by other FORTRAN processors.

The CHr (CHaracter raw) command line switch tells PROMULA.FORTRAN simply to translate FORTRAN character variables into C arrays of `char`. When this switch is active, references to variable length character strings in subprograms cause a syntax error.

The CHs (CHaracter string) command line switch tells PROMULA.FORTRAN to treat all references to character variables as ordered pairs — a pointer and a length. These two values are always shown side-by-side. This allows full use of variable length character strings, but makes the translations more complicated and difficult to maintain.

The CHd (CHaracter dynamic) switch, which is the default, is a more complicated approach than the one above. Though CHs is able to deal with all uses of variable length character strings, it is not able to deal with arbitrary mixtures of character and non-character subprogram arguments. Using CHd, character string lengths are collected at the back of subprogram calls, rather than being paired with their sources directly. In addition, string concatenations are dynamically allocated to allow for arbitrary complexity across subprograms, and substring expressions are calculated via a temporary in some cases. Though CHd conversion does make the relation between FORTRAN source and C output more complex than do the other two switches, its result is highly readable and produces correct results in all cases that we have seen. The CHd approach is sufficiently robust to deal with all situations and is therefore the default setting.

The CHv (CHaracter vector) switch, is by far the most complicated approach. It has all the robustness of the CHd setting. It differs from CHd in that intermediate "descriptor" variables are created which contain the address and length information. The addresses of these intermediate variables are then passed as arguments. This approach has the advantage that there is a one-to-one correspondence between arguments in the FORTRAN and the C. The disadvantage is that additional logic is needed to create the descriptor variables.

In the following subsections a series of examples are presented to show how the four switches CHr, CHs, CHd, and CHv treat FORTRAN character manipulation.

2.5.1 Initializing Character Values

The first example shows how character variables are initialized. It presents a CHARACTER type declaration with associated initializations. Though character initialization is performed in the same manner via the four character manipulation switches, the technique required brings home the difference between the approaches to character strings between FORTRAN and C.

The following FORTRAN code simply initializes a character array `NAME` with the names `Charles`, `Frederick`, `Andrew`, `Mary`, and `Mary` again. Note that the notation used is straightforward, and that a simplified notation can be used to give the last two names the same value.

```
      SUBROUTINE DEMO
      CHARACTER*10 NAME(4:8)/'Charles','Frederick'
     +  ,'Andrew',2*'Mary'/
      WRITE(*,*) NAME
      END
```

The default C representation for this example is shown below.

```
void demo()
{
static char name[50] = {
    'C','h','a','r','l','e','s',' ',' ',' ','F','r','e','d',
    'e','r','i','c','k',' ','A','n','d','r','e','w',' ',' ',
    ' ',' ','M','a','r','y',' ',' ',' ',' ',' ',' ','M','a',
    'r','y',' ',' ',' ',' ',' ',' '
};
    WRITE(OUTPUT,LISTIO,DO,5,STRG,name,10,0);
}
```

Note first that `name` is simply declared as an array of char containing 50 entries. All structure has been reduced to a simple vector. The compiler itself, however, remembers the structure and uses it when necessary.

Note second that the initialization of the `name` array shows up one of the quirks of C. There is no way to specify a sequence of characters without using cumbersome notation, because the notation

```
"Charles"
```

would generate a null-terminated string and the notation

```
'Charles'
```

is not accepted by many compilers, even though it seems completely clear and unambiguous.

Note third in the initializations another point about which more will be said in other sections. The definition of `'Mary'` must be repeated twice, since C has no equivalent of `n*value` notation used in FORTRAN. Note also that FORTRAN character strings are always padded with blanks and not nulls as is the convention in C. It is this fact that makes C and FORTRAN character manipulation completely incompatible. For C then, a FORTRAN CHARACTER variable is treated simply as an array of `char` and not as a C string.

The discussion in this chapter of the Dr (Data runtime) flag presents an alternative approach to the above, in which a small amount of runtime efficiency can be sacrificed to improve the readability of the above.

2.5.2 Subprogram Arguments

The subsection above highlighted that aspect of character manipulation which is common to all four approaches to character manipulation. This subsection describes the major way in which the three approaches differ.

Consider the following simple FORTRAN code fragment which calls the FORTRAN system function `index` and a user written function `jindex`. Both these functions take two character string arguments.

```
      SUBROUTINE DEMO(I,J)
      CHARACTER*12 NAME
      CHARACTER*4 INITIAL
      I = INDEX(NAME,INITIAL)
      J = JINDEX(NAME,INITIAL)
      RETURN
      END
```

The simplest treatment of this code is obtained via the CHr flag. It is as follows:

```
void demo(i,j)
int *i,*j;
{
```

```
extern int jindex();
static char name[12],initial[4];
    *i = fifindex(name,12,initial,4);
    *j = jindex(name,initial);
    return;
}
```

In this version, notice first that the FORTRAN intrinsic function `index` is converted to `fifindex` which is the PROMULA.FORTRAN runtime library function equivalent. In addition, the lengths of the two character strings are passed as well, since `fifindex` needs this information. The treatment of FORTRAN system functions is not affected by the CH flags. Under CHr, the call to the `jindex` subprogram is quite different. Here the subprogram is simply passed pointers to the start of the two character arrays. No length information is given. If `jindex` does not need this information, then the above is the cleanest treatment; however, in most cases it is insufficient.

The treatment under the CHs flag looks as follows:

```
void demo(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
    *i = fifindex(name,12,initial,4);
    *j = jindex(name,12,initial,4);
    return;
}
```

Under this treatment the `jindex` subprogram is treated in exactly the same manner as the `index` function. The lengths of the character strings follows directly after the pointers to their starting points. This treatment works in most cases, but not in the case where character and non-character arguments are both passed to the subprogram.

The CHd result is shown below:

```
void demo(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
    *i = fifindex(name,12,initial,4);
    *j = jindex(name,initial,12,4);
    return;
}
```

This treatment differs from the one under CHr in that character lengths are passed to the subprogram, and it differs from CHs in that the lengths are not directly associated with their pointers; rather, they are collected and placed at the end of the call. Note again that the call to the system function `index` is unaffected.

Finally, the treatment under the CHv flag is as follows:

```
void demo(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
static string T1 = { NULL, 0 };
static string T2 = { NULL, 0 };
    *i = fifindex(name,12,initial,4);
    T1.a = name; T1.n = 12;
    T2.a = initial; T2.n = 4;
    *j = jindex(&T1,&T2);
    return;
```

```
}
```

In this version the temporary variables `T1` and `T2` are introduced to contain the addresses and lengths. Note that the "string" variable type can easily be replaced by a particular "descriptor" type as required by particular platforms. This topic is discussed extensively in the chapter on the configuration file under the topic of "keyword replacement".

More complicated treatments of the character arguments than the CHs treatment are needed because of examples such as the following:

```
      SUBROUTINE DEMO1(I,J)
      CHARACTER NAME*12,INITIAL*4
      J = INDEX(NAME,INITIAL)
      I = JINDEX(NAME,INITIAL)
      RETURN
      END
      SUBROUTINE DEMO2(I,J)
      INTEGER NAME(3),INITIAL
      J = INDEX(NAME,INITIAL)
      I = JINDEX(NAME,INITIAL)
      RETURN
      END
      SUBROUTINE DEMO3(I,J)
      CHARACTER NAME*12
      INTEGER INITIAL
      J = INDEX(NAME,INITIAL)
      I = JINDEX(NAME,INITIAL)
      RETURN
      END
```

In this example, the system function `index` and the user function `jindex` are both called with CHARACTER arguments and with INTEGER arguments in different places in the program. The CHs treatment of the above looks as follows:

```
void demo1(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
    *j = fifindex(name,12,initial,4);
    *i = jindex(name,12,initial,4);
    return;
}
void demo2(i,j)
int *i,*j;
{
extern int jindex();
static int name[3],initial;
    *j = fifindex(name,12,&initial,4);
    *i = jindex(name,&initial);
    return;
}
void demo3(i,j)
int *i,*j;
{
extern int jindex();
static char name[12];
static int initial;
    *j = fifindex(name,12,&initial,4);
    *i = jindex(name,12,&initial);
    return;
}
```

Note that no matter how the `jindex` subprogram is treated, the above cannot hope to execute properly, since the ordering of the arguments is destroyed via the placement of lengths following the character arguments. Again, notice that the `index` system function is unaffected by these games. The CHd result below shows the best approach:

```
void demo1(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
    *j = fifindex(name,12,initial,4);
    *i = jindex(name,initial,12,4);
    return;
}
void demo2(i,j)
int *i,*j;
{
extern int jindex();
static int name[3],initial;
    *j = fifindex(name,12,&initial,4);
    *i = jindex(name,&initial);
    return;
}
void demo3(i,j)
int *i,*j;
{
extern int jindex();
static char name[12];
static int initial;
    *j = fifindex(name,12,&initial,4);
    *i = jindex(name,&initial,12);
    return;
}
```

With the lengths all moved to the back, the calls to `jindex` at least always begin with the two pointers. Remember that we are not dealing only with well-written FORTRAN codes. Things such as the above often occur in "real" FORTRAN programs.

Finally, the listing below shows the translation using the CHv switch.

```
void demo1(i,j)
int *i,*j;
{
extern int jindex();
static char name[12],initial[4];
static string T1 = { NULL, 0 };
static string T2 = { NULL, 0 };
    *j = fifindex(name,12,initial,4);
    T1.a = name; T1.n = 12;
    T2.a = initial; T2.n = 4;
    *i = jindex(&T1,&T2);
    return;
}
void demo2(i,j)
int *i,*j;
{
extern int jindex();
static int name[3],initial;
    *j = fifindex(name,12,&initial,4);
    *i = jindex(name,&initial);
    return;
}
void demo3(i,j)
```

```
int *i,*j;
{
extern int jindex();
static char name[12];
static int initial;
static string T1 = { NULL, 0 };
    *j = fifindex(name,12,&initial,4);
    T1.a = name; T1.n = 12;
    *i = jindex(&T1,&initial);
    return;
}
```

As with CHd, this version at least keeps the argument positions correct.

2.5.3 Substrings

Once the CHd or CHv options are selected, the relation between character variables and their character lengths becomes much more dynamic than with the other two approaches. As a result, various other minor problems associated with FORTRAN character management can be solved. Consider the following code fragment:

```
SUBROUTINE DEMO1
CHARACTER NAME*12,INITIAL*4
INTEGER I,J
INITIAL = NAME(5:8)
INITIAL = NAME(I:J)
INITIAL = NAME(INDEX(NAME,INITIAL):J)
RETURN
END
```

In this example a substring of the `name` variable is being stored in the `initial` variable. In the third statement a complicated expression is used to compute the start of the substring. Under CHr or CHs, the C looks as follows:

```
void demo1()
{
static int i,j;
static char name[12],initial[4];
    ftnsac(initial,4,(name+4),4);
    ftnsac(initial,4,(name+i-1),j-(i-1));
    ftnsac(initial,4,(name+fifindex(name,12,initial,4)-1),
        j-(fifindex(name,12,initial,4)-1));
    return;
}
```

The basic problem is that to use a substring both a starting position and length are needed. The starting position is simply `min-1`; while the length can be computed as `max-(min-1)`. In the simple case where the substring range is constant, PROMULA.FORTRAN performs the computation at compile time. When the range is not constant, the computation must be inserted in the code. This can be seen in the second and third examples. The third example shows the problem. Since the value of `min` is needed twice in the computations, the `min` expression is evaluated twice. This double evaluation is inefficient.

Under the CHd and CHv approach, the descriptor is used. This looks as follows:

```
void demo1()
{
static char name[12],initial[4];
static int i,j;
static string T1;
    ftnscopy(initial,4,(name+4),4,NULL);
    ftnscopy(initial,4,(name+i-1),j-(i-1),NULL);
```

```
    T1.n=fifindex(name,12,initial,4)-1;T1.a=name+T1.n;T1.n=j-T1.n;
    ftnscopy(initial,4,T1.a,T1.n,NULL);
    return;
}
```

Under this approach the two members of the `T1` are computed using the `min` only once. In the following copy request the members of `T1` are then used rather than the source. Notice that CHd and CHv also translate character assignment differently. This issue is discussed in the next subsection under character concatenation.

2.5.4 Character Concatenations

It is from its treatment of string concatenation that CHd (CHaracter dynamic) gets its name. Both CHr and CHs allow for static concatenations but deal only with simple examples involving concatenations done on the fly. Consider the following FORTRAN fragment:

```
      SUBROUTINE DEMO1
      CHARACTER SUM*13,INITIAL*4
      INITIAL = "ONE"
      SUM=INITIAL // "PLUS " // INITIAL
      OPEN(1,FILE="C:"//SUM)
      CALL DEMO2(SUM // "PLUS THREE", SUM // "PLUS TWO")
      RETURN
      END
```

This fragment shows not only a concatenation associated with an assignment, but also concatenations as passed to the `OPEN` operation and as passed as subprogram arguments. The C for this fragment under CHs would be as follows:

```
void demo1()
{
extern void demo2();
static char sum[13],initial[4];
    ftnsac(initial,4,"ONE",3);
    ftnsac(sum,13,ftnads(ftnads(initial,4,"PLUS ",5),9,initial,4),13);
    OPEN(1,FILEN,ftnads("C:",2,sum,13),15,0);
    demo2(ftnads(sum,13,"PLUS THREE",10),23,ftnads(sum,13,"PLUS TWO",8),21);
    return;
}
```

In this code a simple function `ftnads` which returns a pointer to a `char` is used to perform all the concatenations. Each call to `ftnads` adds another string to a static array. In the case of the assignment statement, the temporary result in `ftnads` is copied into the result variable immediately after it is computed. For the `OPEN` and the argument cases, however, the result is left in `ftnads`. For the `OPEN` this does not matter, but in the case of the arguments the second call to `ftnads` either destroys the first argument or concatenates the second to it. Under either case `demo2` will not produce the right result.

The CHd approach is general. String concatenations themselves are simpler and dynamic allocations are allocated memory at runtime. The following is the default CHd C output for the above:

```
void demo1()
{
extern void demo2();
static char sum[13],initial[4];
static string T1,T2;
    ftnscopy(initial,4,"ONE",3,NULL);
    ftnscopy(sum,13,initial,4,"PLUS ",5,initial,4,NULL);
    ftnsallo(&T1,"C:",2,sum,13,NULL);
    OPEN(1,FILEN,T1.a,T1.n,0);
    free(T1.a);
    ftnsallo(&T1,sum,13,"PLUS THREE",10,NULL);
```

```
    ftnsallo(&T2,sum,13,"PLUS TWO",8,NULL);
    demo2(T1.a,T2.a,T1.n,T2.n);
    free(T1.a);
    free(T2.a);
    return;
}
```

In the case of assignment, the `ftnscopy` function replaces the combined effect of `ftnads` and `ftnsac` as presented earlier. The `ftnscopy` function takes a variable number of arguments, terminated by a `NULL`. It concatenates the second and beyond strings directly into the first. In the case of dynamic allocations the `string` type introduced in the previous subsection is combined with a function `ftnsallo`. This function computes the length needed for the concatenation, dynamically allocates memory for it, and then does the copies. The results are stored in the `string` temporary passed as the first argument. This approach is completely general. The only problem is that, once the result of the concatenation is no longer needed, its memory must be freed. This adds a bit of additional code.

The CHv translation of this fragment is as follows:

```
void demo1()
{
extern void demo2();
static char sum[13],initial[4];
static string T1 = { NULL, 0 };
static string T2 = { NULL, 0 };
    ftnscopy(initial,4,"ONE",3,NULL);
    ftnscopy(sum,13,initial,4,"PLUS ",5,initial,4,NULL);
    ftnsallo(&T1,"C:",2,sum,13,NULL);
    OPEN(1,FILEN,T1.a,T1.n,0);
    free(T1.a);
    ftnsallo(&T1,sum,13,"PLUS THREE",10,NULL);
    ftnsallo(&T2,sum,13,"PLUS TWO",8,NULL);
    demo2(&T1,&T2);
    free(T1.a);
    free(T2.a);
    return;
}
```

Notice that the call to `demo2` now receives the descriptor pointers directly. This is the only context in which the CHv formulation is simpler than the CHd one.

2.5.5 Character Treatment Conclusion

The default CHd flag does produce slightly longer translations in some complicated cases. Alternatively, the code it produces is almost always faster. It is strongly recommended that the CHr and CHs flags only be used in special cases where a particularly simple conversion is being performed.

The CHv flag should only be used if external libraries or system functions which assume descriptors must be called. In this case the user can modify the `string` type to reflect the actual required type. See the chapter on the configuration file for more information on this topic.

2.6 Appearance of COMMENTS in C Output — CM0, CM1, CM2

PROMULA FORTRAN translates both embedded and inline comments. Embedded comments are FORTRAN statements with a 'C' or an '*' in column 1. Inline comments usually are placed on the same line as a FORTRAN statement starting to the right of an '!' (VAX FORTRAN convention) or a '/*' (PRIME FORTRAN convention) which indicates where the statement ends and the comment begins.

The comment control switches CM0, CM1, and CM2 simply specify whether comments from the source program should be included in the translation and what form to use if they are included. The CM0 switch excludes comments; the CM1 switch includes them and blocks comments which follow each other, while CM2 includes comments but does not block them. Normally, for the optimized bias CM0 is active; and for the other biases, including the default, CM1 is active.

To see the difference between CM1, which blocks comments, and CM2 which does not, consider the following code.

```
      SUBROUTINE TEST(I,J,K)
C  Compute the value of I using the following relation:
C   J + K - I = 5
      I = J + K - 5
      END
```

The default C output, which uses CM1, produces the following output.

```
void test(i,j,k)
long *i,*j,*k;
{
/*
  Compute the value of I using the following relation:
   J + K - I = 5
*/
    *i = *j+*k-5L;
}
```

In this output, the comments associated with the statement have been grouped and combined into a single C multiline comment. The alternative, using CM2, produces the following:

```
void test(i,j,k)
long *i,*j,*k;
{
/*  Compute the value of I using the following relation:*/
/*   J + K - I = 5*/
    *i = *j+*k-5L;
}
```

In this output, each FORTRAN comment line becomes an equivalent C comment line.

Note that the NCn switch described in another section of this chapter describes the conventions used for inline comments.

2.7 Treatment of DATA Initializations — Da, Dc, Dr

The FORTRAN language has an excellent notation for describing the initial values to be assigned to FORTRAN variables. The following summarizes the FORTRAN syntax:

```
DATA nlist/clist/ [[,]nlist/clist/]...
```

Where:

`nlist` is a list of names to be initially defined. Each name in the list can take one of the forms:

`var`	is a variable name.
`array`	is an array name.
`element`	is an array element name (i.e., subscripted array name.)
`substring`	is a substring of a character variable or array element.

`dolist` is an implied `DO` list of the form:

```
(dlist, i=init,term[,incr])
```

Where:

`dlist` is a list of array element names and implied DO lists. Subscript expressions must consist of integer constants and active control variables from DO list.

`i` is an integer variable called the implied DO variable.

`init` is an integer constant, symbolic constant, or expression specifying the initial value, as for DO loops.

`term` is an integer constant, symbolic constant, or expression specifying the terminal value, as for DO loops.

`incr` is an integer constant, symbolic constant, or expression specifying the increment, as for DO loops.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c, r*c, r(c[,c...]), r((c[,c...]))
```

Where:

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant. The repeat count can repeat the value of a single constant, or can repeat the values of a list of constants enclosed in parentheses. To specify repetition of a complex constant, another set of parentheses must be used.

In addition to DATA statements themselves, the declaration of any variable be it in a DIMENSION statement, or a type statement, or a COMMON statement may be followed by a set of initialization values enclosed in slashes.

The biggest problem in the translation of initializations from FORTRAN to C relates to a limitation in C syntax. Simply stated, there is no equivalent of the n*value notation in FORTRAN. Thus, if you need to initialize a large array with a single value, you must write that value over and over again. The other problem has to do with initializing a numeric variable with a character constant, which C does not allow in its initializations.

The default setting for PROMULA FORTRAN uses the C static variable initialization notation for FORTRAN DATA initialization of local variables. This is the most efficient method and works for most cases. The alternative is to use a combination of FORTRAN READ NAMELIST and internal file capabilities to set the data initialization values at runtime. This method is less efficient because it requires the construction of tables in the code, and because it must be executed. This method is always used for initializations to COMMON and when you use the DR (data runtime) switch on the command line.

2.7.1 Overview of Initialization Problem

The following example shows a set of data initializations which highlight several important points about DATA statements in FORTRAN.

```
      SUBROUTINE DEMO
      DIMENSION B(5), X(5,5), C(10)
      DATA A/4.2/,B(1)/5.4/
      DATA ((X(J,I),I=1,J),J=1,5)/1,2,2,3*3,4*4,5*5/
      DATA J/4/,(C(J),J=1,10)/10*2.0/
    1 FORMAT(1X,10F10.1)
    2 FORMAT(1X,I10)
      WRITE(*,1) A, B
      WRITE(*,1) C
      WRITE(*,2) J
```

```
      DO 10 J = 1,5
      WRITE(*,1) (X(J,I),I=1,5)
   10 CONTINUE
      STOP
      END
```

Note first in the initialization of `X` that the value of the dummy subscript `J` is used as a terminal point for the subscript `I`. This is allowed in DATA statements. Note next that, in the following DATA statement, a program variable `J` is given the value of 4, and then a dummy variable `J` in the DATA statement is allowed to run from 1 to 10. Dummy variables in DATA statements are not program variables and are independent of them.

The listing below shows the default translation of the above example.

```
void demo()
{
static float b[5] = {
    5.4,0.0,0.0,0.0,0.0
};
static float x[5][5] = {
    1.0,2.0,3.0,4.0,5.0,0.0,2.0,3.0,4.0,5.0,0.0,0.0,
    3.0,4.0,5.0,0.0,0.0,0.0,4.0,5.0,0.0,0.0,0.0,0.0,5.0
};
static float c[10] = {
    2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0,2.0
};
static float a = 4.2;
static long j = 4;
static long i;
static char* F1[] = {
    "(1x,10f10.1)"
};
static char* F2[] = {
    "(1x,i10)"
};
    WRITE(OUTPUT,FMT,F1,1,REAL4,a,DO,5,REAL4,b,0);
    WRITE(OUTPUT,FMT,F1,1,DO,10,REAL4,c,0);
    WRITE(OUTPUT,FMT,F2,1,INT4,j,0);
    for(j=1L; j<=5L; j++) {
        WRITE(OUTPUT,FMT,F1,1,MORE);
        for(i=0L; i<5L; i++) {
            WRITE(REAL4,x[i][j-1],MORE);
        }
        WRITE(0);
    }
    STOP(NULL);
}
```

Note that values not explicitly defined in the DATA statement are given a value of 0, blanks for character strings. Note also that the initializations of `X` and `J` are correct.

The following listing shows the same translation using the DR command line switch.

```
void demo()
{
static float b[5];
static float x[5][5];
static float c[10];
static float a;
static long j;
static long i;
static int ftnsiz[] = {1,1,5,1,1,25,1,1,10};
```

```
static namelist DATAVAR[] = {
"b",b,6,ftnsiz,"x",x,6,ftnsiz+3,"c",c,6,ftnsiz+6,
"a",&a,6,NULL,"j",&j,5,NULL
};
static char *DATAVAL[] = {
"$DATAVAR",
"b=5.4,4*0.0,x=1.0,2.0,3.0,4.0,5.0,0.0,2.0,3.0,4.0,"
"5.0,2*0.0,3.0,4.0,5.0,",
"3*0.0,4.0,5.0,4*0.0,5.0,c=10*2.0,a=4.2,j=4,",
"$END"
};
static FIRST = 1;
static char* F1[] = {
    "(1x,10f10.1)"
};
static char* F2[] = {
    "(1x,i10)"
};
    if(FIRST) {
        FIRST=0;
        fiointu((char*)DATAVAL,0,2);
        fiornl(DATAVAR,5,NULL);
    }
    WRITE(OUTPUT,FMT,F1,1,REAL4,a,DO,5,REAL4,b,0);
    WRITE(OUTPUT,FMT,F1,1,DO,10,REAL4,c,0);
    WRITE(OUTPUT,FMT,F2,1,INT4,j,0);
    for(j=1L; j<=5L; j++) {
        WRITE(OUTPUT,FMT,F1,1,MORE);
        for(i=0L; i<5L; i++) {
            WRITE(REAL4,x[i][j-1],MORE);
        }
        WRITE(0);
    }
    STOP(NULL);
}
```

See the discussion of NAMELIST in the FORTRAN Compiler User's Manual for more information on namelist. In essence, to do the data initialization at runtime, PROMULA FORTRAN invents a NAMELIST which contains all of the variables involved in data initializations. This is done in the initialization of `ftnsiz` which contains the array bound specifications, and in the initialization of the structure DATAVAR which contains the namelist specification. Next, the data quantity lists and values are converted into namelist read input format and stored in an internal character file called DATAVAL. Finally, a variable FIRST is introduced which triggers the calls to the runtime processing functions needed to initialize the actual data values. NAMELIST is a much neglected and maligned feature of FORTRAN which solves this particular problem very nicely.

2.7.2 The Initialization Switches

As is discussed above, C is needlessly weak in the area of data initialization. Since we cannot change C, we must accommodate this weakness in our C output conventions. The D command line switch specifies when data initializations are to be performed and what storage type initialized variables should have. Its individual settings are as follows:

Setting	Meaning
Dc	This is the default setting, despite its problems. It requests that initialization produced by the DATA statement to local variables be implemented at compile time. The initialization values are actually placed in the C source code.

Dr	This setting requests that data initializations be copied from the DATA statement into NAMELIST form into an internal character file so that they can be read into the actual variables at runtime.
Da	This setting modifies the behavior of the Dc flag by specifying that variables initialized at compile time should be declared as being "auto" as opposed to the default of "static".

A summary example is included here to show the effects of this switch. The following program contains several data initializations.

```
      PROGRAM DEMO
      DIMENSION A(50)/10*0.0,40*1.0/
      CHARACTER NAME*4(4)/4HFred,2*'Mary',3HJoe/
      CALL DEMO1(A)
      STOP
      END
      SUBROUTINE DEMO1(A)
      DIMENSION A(50)
      DATA VAL/55.6/
      DO 10 I = 1,50
10    A(I) = A(I) * val
      RETURN
      END
```

These initializations are performed either as part of the declaration or in an explicit DATA statement. The D switch treats both types of initializations in the same manner.

The following is the default translation of the example above.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void demo1();
static float a[50] = {
    0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    1.0,1.0
};
static char name[16] = {
    'F','r','e','d','M','a','r','y','M','a','r','y',
    'J','o','e',' '
};
    ftnini(argc,argv,NULL);
    demo1(a);
    STOP(NULL);
}
void demo1(a)
float a[];
{
static float val = 55.6;
static long i;
    for(i=0L; i<50L; i++) {
        a[i] = a[i]*val;
    }
    return;
}
```

The two primary problems to note are that C has no equivalent of the `n*value` notation and that a non-null terminated sequence of characters can only be written as individual elements enclosed in single quotes. Nonetheless, this way of initializing variables is much more efficient than the runtime way and is the default.

The following shows the same translation with the Dr switch set.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void demo1();
static float a[50];
static char name[16];
static int ftnsiz[] = {1,1,50,1,1,4};
static namelist DATAVAR[] = {
"a",a,6,ftnsiz,"name",name,16,ftnsiz+3
};
static char *DATAVAL[] = {
"$DATAVAR",
"a=10*0.0,40*1.0,name='Fred',2*'Mary','Joe',",
"$END"
};
    ftnini(argc,argv,NULL);
    fiointu((char*)DATAVAL,0,2);
    fiornl(DATAVAR,2,NULL);
    demo1(a);
    STOP(NULL);
}
void demo1(a)
float a[];
{
static float val;
static long i;
static namelist DATAVAR[] = {
"val",&val,6,NULL
};
static char *DATAVAL[] = {
"$DATAVAR",
"val=55.6,",
"$END"
};
static FIRST = 1;
    if(FIRST) {
        FIRST=0;
        fiointu((char*)DATAVAL,0,2);
        fiornl(DATAVAR,1,NULL);
    }
    for(i=0L; i<50L; i++) {
        a[i] = a[i]*val;
    }
    return;
}
```

In this translation PROMULA FORTRAN has constructed logic and simple static initializations which allow the variable values to be read into the actual variables at runtime. Using this technique, the value strings can be kept simple. Also, any problems having to do with weak-typing or hiding characters are easily solved. This runtime initialization gives us all the flexibility we need, but it does require that extra code be linked with the program, and that the value be stored "twice".

Finally, the listing below shows the C output when using the Da switch:

```
void main(argc,argv)
```

```
int argc;
char* argv[];
{
extern void demo1();
auto float a[50] = {
    0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
    1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0
};
auto char name[16] = {
    'F','r','e','d','M','a','r','y','M','a','r','y',
    'J','o','e',' '
};
    ftnini(argc,argv,NULL);
    demo1(a);
    STOP(NULL);
}
void demo1(a)
float a[];
{
auto float val = 55.6;
static long i;
    for(i=0L; i<50L; i++) {
        a[i] = a[i]*val;
    }
    return;
}
```

This version declares the initialized variables as `auto`. Note that many C compilers are unable to compile this version. The effect of this auto declaration is that the variables are initialized at their specified values each time the function containing them is entered.

2.8 Turn on Debugging Mode — DB

When using PROMULA.FORTRAN as a compiler it is often convenient to use a debugger. The DB switch makes this possible. When used, the debugger will refer to the original FORTRAN source code rather than to the intermediate C code. This flag is actually only a macro for the following flag settings:

CM0:	Turn comments off
Ln:	Include line number information
L0:	Do not generate newlines within statement output
Qe32000:	Allow 32000 bytes for line number information

These flags are discussed in detail elsewhere in this chapter.

2.9 Echo Control Options — ES, ET, EX, EZ, EP, EL

Except for its initial banner, PROMULA.FORTRAN is normally silent, unless a fatal error is encountered. Messages are sent to standard output and may be redirected or may be sent to a listing file (see section on the PN and PA switches). Note that error messages themselves are discussed in the PROMULA FORTRAN Compiler manual. The echo control options may be used to send additional information to the listing output. This information includes the following:

Option	Information sent to listing output
EL1	Warnings about potentially serious inconsistencies or usages in the FORTRAN source code
EL2	Comments about possibly non-portable code or about code that may be incorrect and the warnings from above

EL3	Notes about standard FORTRAN violations and other miscellaneous observations and comments and warnings from above.
EP	A listing of the intermediate pseudo-code produced by the compiler.
ES	An annotated listing of the source code.
ET	An annotated listing of the C output code.
EX	An alphabetical listing of the symbols used in the source along with a summary description of each and a cross-reference listing of symbol references by line number.
EZ	A listing of the intermediate symbol information produced by the compiler.

2.9.1 Warnings, Notes, and Comments

The form and meaning of the warnings, notes, and comments are discussed in the PROMULA FORTRAN Compiler manual in its chapter on error messages. Suffice to say that PROMULA.FORTRAN does extensive syntax checks while it is processing the source code and extensive consistency checks after it has processed each subprogram.

The following is a sample listing produced which shows the type of messages that might be produced at the EL1 level:

390: utest.for: W818: The argument "ia" is being defined with type integer*4 when it has been passed an argument of type character.

The "W" appended to the error number indicates a warning. Messages at this level can be ignored, but they typically indicate potentially serious problems.

At the EL2 level the following additional types of messages are issued:

54: utest.for: C870: The array OBUF is being subscripted with less than 1 expressions.
380: utest.for: C815: A data value of type character is being assigned to the variable IA of type integer*4.
520: utest.for: C816: The binary type character is being used where type integer*2 is expected.
558: utest.for: C861: Data is being allocated to common storage via the variable SEATRD.
820: utest.for: C866: The real*4 type has previously been assigned to UC_.

At this level in addition to warnings other usages are isolated that should either be checked or that represent potentially serious portation problems.

Finally, at the EL3 level the following additional types of messages are issued:

37: utest.for: N858: The identifier THERMA1 with more than 6 characters is nonstandard.
184: utest.for: N864: Declarative statements following executable statements is nonstandard.
223: utest.for: N851: The use of inline comments is nonstandard.
295: utest.for: N872: Equivalencing CBUF of type character*80 with IBUF of type INTEGER*2 is nonstandard.
343: utest.for: N853: The standard delimiter for a character constant is the single quote.
431: utest.for: N858: The identifier PRANDOM_INDX$ with more than 6 characters is nonstandard.
474: utest.for: N806: Omitting the comma after the I FORMAT specification is nonstandard.
545: utest.for: N823: The COMMON block DIR1 has character*1 variable VFORMS and an unspecified variable OPTEV.
722: utest.for: N801: The INCLUDE statement is nonstandard.
1: STRUC6.INC: N801: The STRUCTURE statement is nonstandard.
800: utest.for: N833: The non-parenthetical form of the PARAMETER statement is nonstandard.

As can be seen, the EL3 level generates many messages and is primarily intended for those who are trying to write pure standard conforming code.

2.9.2 Annotated Listing of Source Code

The following listing was produced via the ES option for a simple subprogram referencing a single include file:

```
PROMULA FORTRAN Compiler V4.00       Date: 08/11/92  Time: 09:35  Page: 1
File: utest.for

If Line# Nl Source
-- ----- -- ------
       1           SUBROUTINE STRUC6
       2           INCLUDE 'STRUC6.INC'
 1     1           STRUCTURE /DATE/
 1     2  1            INTEGER*4 DAY,MONTH
 1     3  1            INTEGER*4 YEAR
 1     4  1        END STRUCTURE
 1     5    C
 1     6    C           STRUCTURE /TIME/ APP_TIME(2)
 1     7    C               LOGICAL*1 HOUR,MINUTE
 1     8    C           END STRUCTURE
 1     9    C
 1    10    C      This is the same as:
 1    11    C
 1    12    C           STRUCTURE /TIME/
 1    13    C               LOGICAL*1 HOUR,MINUTE
 1    14    C           END STRUCTURE
 1    15    C           RECORD /TIME/ APP_TIME(2)
 1    16    C
 1    17              STRUCTURE /TIME/
 1    18  1               LOGICAL*1 HOUR,MINUTE
 1    19  1           END STRUCTURE
 1    20
 1    21           STRUCTURE /APPOINTMENT/
 1    22  1           RECORD /DATE/ APP_DATE
 1    23  1
 1    24  1           RECORD /TIME/ APP_TIME(2)
 1    25  1
 1    26  1           CHARACTER*20 APP_MEMO(4)
 1    27  1           LOGICAL*1 APP_FLAG
 1    28  1       END STRUCTURE
 1    29           RECORD /APPOINTMENT/ NEXT_APP,APP_LIST(7)
 1    30           RECORD /DATE/ TODAY
       3           WRITE(6,*) '********* STRUC6.OUT'
       4           DO 10 I = 1,7
       5  1            CALL GET_DATE(I,TODAY)
       6  1            WRITE(6,*) TODAY.DAY,TODAY.MONTH,TODAY.YEAR
       7  1            APP_LIST(I).APP_DATE = TODAY
       8  1            TODAY.DAY = TODAY.DAY + 1
       9  1   10   END DO
      10       5   FORMAT(3I5)
      11           NEXT_APP = APP_LIST(1)
      12           OPEN(1,FILE='STRUC6.BIN',FORM ='UNFORMATTED',STATUS='UNKNOWN')
      13           WRITE(1) NEXT_APP
      14           WRITE(1) APP_LIST
      15           END
```

The heading which is printed at the top of each page contains the name of this compiler along with its current version number on the left-hand-side of the page. The right-hand-side normally contains the name of the file being compiled, the date, the time, and the page number relative to the file. In this case the page width was set to be narrow, so the name of the file is placed on a second line.

The annotated listing itself contains the include file number (If), the line number within the source file, the nesting level of the statement, and the actual source code statement. The include file number is left blank for statements in the original source file. The nesting level indicator is used with declaration statements when structures are being defined. It indicates the level of nesting within the structure. For executable statements the nesting level indicator indicates the degree of nesting within DO and/or IF statements. If there is no current nesting then the nesting level is left blank.

2.9.3 Symbol Listing and Cross Reference Table

The following symbol listing and cross reference table was produced using the Ex option. The table consists of 4 sections: Include files, symbols referenced, symbol references by line number, and statement label types and references. In general, in these tables all user defined symbols are shown in uppercase, while all descriptive symbols are shown in lowercase.

The include files section simply lists all include files encountered to date in the compilation along with the sequence number assigned to them. Note that the base source file has a number of zero. If there are no include files referenced in the current subprogram, then this section is omitted.

The philosophy behind the design of the symbols referenced table is that the user will use this table when he wants information about a particular symbol, whose status he may not be familiar with but whose identifier he knows or has seen somewhere in the listing. The report consists of a single alphabetical list of each symbol referenced in the subprogram. For each symbol its object type, its storage status, its binary type, and a comment are provided.

The object type is straightforward. There are thirteen possible entries: `constant`, `parameter`, `variable`, `subroutine`, `function`, `intrinsic`, `namelist`, `entry`, `statefunc`, `structure`, `record`, `pointer`, and `common`. These names correspond directly to the possible FORTRAN object types. It should be mentioned that members within structure definitions are treated simply as variables or records. This convention is compatible with the approach of using a single alphabetized list of all symbols.

The storage status of a symbol can be one of four different things. For subprogram arguments it is specified as `argument`. For variables in common blocks, it is the name of the common block containing the variable. For members of structures, it is the name of structure containing the member. For simple variables it is one of the following: `static`, `auto`, `dynamic`, or `virtual`. See the Sa and Ss switches for a description of static versus auto storage. Dynamic and virtual variables can be created via the PROMULA interface described in the PROMULA FORTRAN Compiler manual.

The type of a variable is simply its type specification. For records it is the structure type of the record.
The comment associated with the symbol is a function of its object type. For constant integer parameters — i.e, those that may be used in other declaration statements — the comment specifies the value of the parameter as specified or computed. For variables or records, the dimensionality is given and for arrays the total size in bytes. For subprograms the number of arguments is given along with the assumed type of each argument. Note that in the C tradition, PROMULA.FORTRAN makes extensive use of subprogram prototypes and always makes certain that argument types are consistent. If they are not it issues a warning.

The symbol reference by line number table is simply that. Along with the line number a use type code is also specified — 'D' means 'defined', 'M' means 'modified', 'U' means 'used', and 'P' means 'passed to a subprogram'. If a symbol has multiple references in a single statement, then only one reference is reported.

Note that if include files are involved, then the line number is followed by the include file sequence number. If include files are not involved, then a simple sequence number is used.

```
PROMULA FORTRAN Compiler V4.00       Date: 08/11/92  Time: 09:35  Page: 2
File: utest.for

Include files used in unit:

Seq Filename
--- --------
  1 STRUC6.INC
```

```
Symbols referenced in SUBROUTINE STRUC6

Identifier       Object     Storage         Type             Comment
----------       ------     -------         ----             -------
APP_DATE         record     APPOINTMENT     DATE             scalar
APP_FLAG         variable   APPOINTMENT     logical*1        scalar
APP_LIST         record     static          APPOINTMENT      1d-array(679)
APP_MEMO         variable   APPOINTMENT     character*20     1d-array(80)
APP_TIME         record     APPOINTMENT     TIME             1d-array(4)
APPOINTMENT      structure
DATE             structure
DAY              variable   DATE            integer*4        scalar
GET_DATE         subroutine                                  2 args(integer*4,
                                                             unspecified)
HOUR             variable   TIME            logical*1        scalar
I                variable   static          integer*4        scalar
MINUTE           variable   TIME            logical*1        scalar
MONTH            variable   DATE            integer*4        scalar
NEXT_APP         record     static          APPOINTMENT      scalar
TIME             structure
TODAY            record     static          DATE             scalar
YEAR             variable   DATE            integer*4        scalar

Symbol references by line number in SUBROUTINE STRUC6

Identifier           Line.If:u (D=defined, M=modified, U=used, P=passed)
----------           --------- ---------- ---------- ------- ---------
APP_DATE               22.01:D       7.00:U
APP_FLAG               27.01:D
APP_LIST               29.01:D       7.00:M      11.00:U      14.00:U
APP_MEMO               26.01:D
APP_TIME               24.01:D
APPOINTMENT            21.01:D
DATE                    1.01:D
DAY                     2.01:D       6.00:U       8.00:U
GET_DATE                5.00:U
HOUR                   18.01:D
I                       4.00:M       5.00:P       7.00:U
MINUTE                 18.01:D
MONTH                   2.01:D       6.00:U
NEXT_APP               29.01:D      11.00:M      13.00:U
TIME                   17.01:D
TODAY                  30.00:D       5.00:P       6.00:U       7.00:U
                        8.00:M
YEAR                    3.01:D       6.00:U

Statement label types and references by line number in SUBROUTINE STRUC6

Label  Type          Line.If:u (D=defined, U=used, A=assigned)
-----  ----          --------- ---------- ------- -----------
5      format          10.00:D
10     statement        4.00:U       9.00:D
```

The statement label types and references table is a numerical listing of the statement labels, along with their type, `statement` or `format`, and a listing of the lines where they are referenced.

If the subprogram contains EQUIVALENCE statements, then a fifth table type is generated: the equivalence pairs table. Given the code fragment below:

```
48          SUBROUTINE ANA1
49          INTEGER BUF1(2048), BUF2(2048), BUF3(2048)
50          BYTE OBUF(32767)
51          EQUIVALENCE (BUF2(1), OBUF(1))
52          EQUIVALENCE (BUF3, OBUF)
62          END
```

The following equivalence pairs table is produced:

```
Equivalence pairs:

Dependent        Base               Offset
---------        ----               ------
BUF2             OBUF                    0
BUF3             OBUF                    0
```

The base variable is the variable whose storage is being used to contain the dependent variable. The offset is the byte offset within the base variable of the start of the dependent variable.

2.9.4 Intermediate Compiler Tables

The EP and EZ switches can be used to generate a listing of the intermediate compiler tables — EZ lists the symbol tables, while EP lists the pseudo-code generated. The following is a simple program along with the listing formed by the EP and EZ switches.

```
      PROGRAM TEST
      INTEGER I,J,K
      PRINT *, 'I+J/K = ',(I+J)/K
      STOP
      END
```

```
Include files used in unit:

Seq Filename
--- --------

Symbols defined in unit:

Base  Identifier  Obj Flag Inf2 Incf  Typ    Ref Normal  Cref Tflags        Adr    Valsize
----  ----------  --- ---- ---- ----  ---    --- ------  ---- ------        ---    -------
278   I             1    0    0    0    5      0      0     1     18          0          4
286   J             1    0    0    0    5      0      0     1     18          0          4
294   K             1    0    0    0    5      0      0     1     18          0          4

Pcode generated by unit:

Location        Operation
--------        ---------
00000           nop
00001           sto
00002           lst
00003           lsc     "I+J/K = ", 8
00006           wrv     17
00009           lda     i, 278, 1
00012           ldr     i, 278, 1
00015           lda     j, 286, 1
00018           ldr     j, 286, 1
00021           adi
00022           lda     k, 294, 1
00025           ldr     k, 294, 1
00028           dvl
00029           wrv     5
00032           wln
00033           ner
00034           nop
00035           stn
00036           nop
```

```
00037           eop
```

These tables show the detailed information generated to form the C translation. A detailed discussion of the actual symbol conventions and pseudo-code is beyond the scope of this manual. Individuals desiring additional information may contact Great Migrations LLC.

2.9.5 Annotated Listing of C Output

In addition to the annotated listing of the source code, a similar listing can also be obtained for the C output produced. The following is the listing produced with the ET flag during the processing of the code used earlier.

```
      If Line# Nl Translation
      -- ----- -- -----------
             1    #define SPROTOTYPE
             2    #include "fortran.h"
             3    void struc6()
             4    {
             5    typedef struct {
             6        long day,month,year;
             7    } Sdate;
             8    typedef struct {
             9        unsigned char hour,minute;
            10    } Stime;
            11    typedef struct {
            12            Sdate  app_date;Stime  app_time[2];char  app_memo[80];unsigned  char
      app_flag;
            13    } Sappointment;
            14    /*
            15              STRUCTURE /TIME/ APP_TIME(2)
            16                  LOGICAL*1 HOUR,MINUTE
            17              END STRUCTURE
            18
            19           This is the same as:
            20
            21              STRUCTURE /TIME/
            22                  LOGICAL*1 HOUR,MINUTE
            23              END STRUCTURE
            24              RECORD /TIME/ APP_TIME(2)
            25    */
            26    extern void get_date();
            27    static Sappointment next_app,app_list[7];
            28    static Sdate today;
            29    static long i,D2;
            30        WRITE(6,LISTIO,STRG,"********* STRUC6.OUT",20,0);
            31        for(i=1,D2=(7-i+1); D2>0; D2--,i+=1) {
            32  1         get_date(&i,&today);
            33                                                                              1
      WRITE(6,LISTIO,INT4,today.day,INT4,today.month,INT4,today.year,0);
            34  1         app_list[i-1].app_date = today;
            35  1         today.day = today.day+1L;
            36        }
            37        next_app = app_list[0];
            38        OPEN(1,FILEN,"STRUC6.BIN",10,STATUS,"UNKNOWN",FORM,"UNFORMATTED",0);
            39        WRITE(1,BYTE,&next_app,(int)(sizeof(Sappointment)),0);
            40        WRITE(1,BYTE,app_list,(int)(7*sizeof(Sappointment)),0);
            41    }
```

2.10 Treatment of Syntax Errors — ER0, ER1, ER2, ER3, ER4

PROMULA.FORTRAN is like any other FORTRAN compiler in that in order to translate the source code, it must also validate it — i.e., check it for any syntax errors. Syntax errors may arise for a variety of reasons:

(1) An actual syntax error

(2) Use of a dialectal feature not supported by the currently active language definition file.

(3) An improperly used command line switch.

(4) A problem with a subprogram parameter.

The basic assumption in the design of PROMULA.FORTRAN is that existing, valid FORTRAN programs are being processed; therefore, a syntax error means that a mismatch of some sort exists between PROMULA.FORTRAN and the dialect being processed. By default, therefore, PROMULA.FORTRAN stops processing when a syntax error occurs. The switch ER0 specifies this behavior. The switches ER1 through ER4 specify that processing should continue despite the syntax error.

With PROMULA.FORTRAN, if processing is to continue despite an error then PROMULA.FORTRAN must be told how to produce the C output given a statement in the source program which it cannot translate. The ER1 through ER4 switches differ in how the error statement appears in the C output. Consider the following simple program which contains an illegal character in its second statement.

```
PROGRAM TEST
I = J @ K
STOP
END
```

The ER1 switch specifies that syntax errors block at C compilation time. In other words, when a syntax error is encountered a message is issued, but processing continues. The statement producing the error will be entered into the C output in the form:

```
SYNTAX ERROR: errno, statement
```

Thus, the program above produces the following C code under ER1:

```
void main(argc,argv)
int argc;
char* argv[];
{
static long i,j;
    ftnini(argc,argv,NULL);
    SYNTAX ERROR: 145,"I = J @ K"
    STOP(NULL);
}
```

Any attempt to compile the output will cause errors at that time. This setting allows you to see all your errors at one time, but blocks you from actually using the results.

The ER2 switch specifies that syntax errors block at link time. It behaves exactly like ER1 above, except that the following is entered into the C output:

```
SYNTAXERROR(errno,"statement");
```

Thus, the program above produces the following C code under ER2:

```
void main(argc,argv)
int argc;
char* argv[];
```

```
{
static long i,j;
    ftnini(argc,argv,NULL);
    SYNTAXERROR(145,"I = J @ K");
    STOP(NULL);
}
```

This setting allows you to process all errors at once and to go ahead with a provisional compilation; however, the unsatisfied external SYNTAXERROR will block any attempt to link the program.

The ER3 switch specifies that syntax errors cause a runtime error to be issued. This switch behaves exactly like ER1 and ER2 above, except that the following is entered into the C output:

```
puts("ERROR errno statement");
```

Thus, the program above produces the following C code under ER3:

```
void main(argc,argv)
int argc;
char* argv[];
{
static long i,j;
    ftnini(argc,argv,NULL);
    puts("ERROR 145 I = J @ K");
    STOP(NULL);
}
```

This setting allows you to process all errors at once, and to go ahead and form an executable; however, when you execute it you will get error messages when the offending statements are encountered.

Finally, the ER4 switch specifies that syntax errors cause warnings only. This switch is like the ones above except that the following is entered into the C output:

```
 /* SYNTAX ERROR: errno, statement */
```

Thus, the program above produces the following C code under ER4:

```
void main(argc,argv)
int argc;
char* argv[];
{
static long i,j;
    ftnini(argc,argv,NULL);
/* SYNTAX ERROR: 145,"I = J @ K" */
    STOP(NULL);
}
```

Obviously, this setting treats errors simply as warnings; however, the resultant executable will not behave correctly.

2.11 FORTRAN Input Format Used — Fsnum, Ft, Ff, Fv, F9

In the good old days the one thing that was always the same was the basic line format used to enter FORTRAN programs. But all good things must end. Now there are at least 5 different major variations of the FORTRAN entry format that we know of. The F command line switch allows you to specify the entry format that you are using. There is an extensive discussion of the different formats in the PROMULA FORTRAN Compiler Manual. That discussion will not be repeated here. The individual settings associated with this flag are mutually exclusive and are as follows:

Fsnum Selects the standard fixed format with an ending column of n. The default setting is Fs72, which is that good-old format referred to above.

Ft	Selects tab format which comes from the VAX FORTRANs.
Ff	Selects the free-form format which is relatively typical of those FORTRANs that accepted "terminal input".
Fv	Selects the VS FORTRAN free-form format.
F9	Selects the Fortran 90 format

2.12 Source FORTRAN Integer Type — FIs, FIl

There is variation between FORTRAN compilers as to whether the default type of the INTEGER specification should be INTEGER*2 or INTEGER*4. In fact, an interesting aspect of many modern FORTRAN compilers is that the user may specify whether the default integer type is to be a short 16 bit representation or a long 32 bit representation on the command line.

The FIs and FIl command line switches in PROMULA.FORTRAN allow for this specification. The FIs specification says that the default integer type is short, INTEGER*2; while FIl specifies that it is long, INTEGER*4. The default setting for this switch is FIl for the standard FORTRAN dialect.

2.13 Gname — Name of File Containing Global Symbols

The PROMULA application management system and PROMULA.FORTRAN can be used in tandem to upgrade existing codes. The interface between these two systems truly adds value to existing FORTRAN programs.

The global symbols file, whose default extension is 'glb' contains a list of program variables that are to be made "global" for use by the PROMULA Application Development System. The content of this file and the general topic of the PROMULA interface are described in detail in the PROMULA FORTRAN Compiler Manual.

2.14 Common Variables Convention — Ga, Gd, Gp, Gs, Gr, Gv

That aspect of FORTRAN which has the highest likelihood of causing translation errors and/or user readability objections is the COMMON statement. Every storage trick and weak-typing trick imagined gets used in the nuances of changing COMMON block definitions through a large program. There is no ideal way to deal with common blocks. PROMULA.FORTRAN translates common blocks in one of six ways. These are controlled via the G command switch. The individual settings associated with this switch are mutually exclusive and are as follows:

Set	Meaning
Gp	Define common blocks via local pointers and varying typedefs. This is the default setting. In particular, COMMON block identifiers are declared simply as external char storage areas. Locally, the address of that storage area is assigned to a structure pointer whose members are defined in the same manner as the COMMON definition in the subprogram being translated. This technique works in all cases except where byte alignment adjustments are needed. The problem with this technique is that it is ugly and that it must use expressions of the form `Cblock->var` to refer to members in the block. These expressions require pointer arithmetic at run-time and, therefore, produce inefficient code, especially on the PC when "large" or "huge" memory models are in effect.
Gd	This is a variation of the Gp option, except that it may produce better code for some environments. Under Gp local auto variables are introduced as structure pointers; while under Gd the same syntactic effect is achieved via a defined symbol.

Gs Defines COMMON blocks as static structures with varying internal composition. In particular, this technique defines each occurrence of the COMMON block as an external structure whose members are defined in the same manner as they are defined in the subprogram being translated. This technique produces readable code and generates relatively efficient code. The problem is that many compilers consider redefining an external with a detailed structure repeatedly in different functions to be a typing error. Some give warnings and others consider this a fatal error. If this technique works with your compiler, and if it does not offend your own view of "symbol typing", then this technique is highly recommended. Note that it can be used freely with the Om flag discussed elsewhere.

Gr Defines COMMON blocks as raw static vectors of char. This technique is used when an exact block layout is required or when the efficiency of the Gs technique is desired in an environment where it cannot be used. Here, the variable positions within the block are calculated using alignment and wordsize specifications that are supplied via the dialect definition. Actual variable references then become references to the COMMON block name plus the calculated position. The COMMON variable identifiers within the block disappear. The code generated is efficient; however, if you intend to maintain the code in its C form, it is difficult to read. This technique is highly recommended for those who are using PROMULA.FORTRAN simply as a preprocessor to their C compiler, with no real interest in the intermediate C output.

Gv Defines COMMON variables as independent external symbols. This technique should only be used when you are building a function library or when all COMMON blocks are always defined in precisely the same way. This setting removes the COMMON blocks from the translation; thus, it is the inverse of the technique above which removes the variables within the blocks. With this technique the variables themselves become external symbols. If appropriate, this setting produces very readable and clean code; however, if used in the wrong context it can produce hash.

Ga This treatment assumes that the user has himself allocated or assigned memory for the common areas; thus, the COMMON variable itself becomes a structure pointer as opposed to a structure, as assumed by all other treatments. The COMMON symbol itself is defined as an instantiated pointer to that structure; thus, giving a very clean looking translation.

Examples of the use of this flag are presented here. The following FORTRAN subroutine contains two COMMON blocks.

```
SUBROUTINE DEMO
CHARACTER C1*15,C2*10
COMMON/ALPHA/A(10,10),B(5),C,D
COMMON/BETA/C1(10),C2(5)
A(1,1) = C
B(5) = D
C1(1)(3:12) = C2(4)
RETURN
END
```

Below is the default translation for this example which uses the Gp setting.

```
void demo()
{
extern char Xalpha[],Xbeta[];
typedef struct {
    float a[10][10],b[5],c,d;
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
typedef struct {
    char c1[150],c2[50];
} Cbeta;
auto Cbeta *Tbeta = (Cbeta*) Xbeta;
    Talpha->a[0][0] = Talpha->c;
```

```
    Talpha->b[4] = Talpha->d;
    ftnscopy((Tbeta->c1+2),10,(Tbeta->c2+30),10,NULL);
    return;
}
char Xalpha[428];
char Xbeta[200];
```

In the function itself, the COMMON blocks `ALPHA` and `BETA` are defined simply as arrays of `char`, note NOT pointers to char. Next there are typedefs whose names are constructed by appending a C to the block name which defines the local structure of the block. Next there is an auto pointer defined, name constructed by adding a T to the block name, which points to the start of the char storage area. This double definition with a typedef and then a variable of that type is needed because the assignment of the address of the common storage area requires a cast to avoid a compiler error or warning. As was said above, the big problem with this translation technique is that it must use the `->` operator. This operator does pointer arithmetic, which on the PC can be relatively inefficient with some memory models. Finally, the storage areas for the blocks themselves are reserved as global storage areas with the maximum size defined for each block.

The translation below is produced via the Gd setting.

```
void demo()
{
extern char Xalpha[],Xbeta[];
typedef struct {
    float a[10][10],b[5],c,d;
} Calpha;
#define Talpha ((Calpha*) Xalpha)
typedef struct {
    char c1[150],c2[50];
} Cbeta;
#define Tbeta ((Cbeta*) Xbeta)
    Talpha->a[0][0] = Talpha->c;
    Talpha->b[4] = Talpha->d;
    ftnscopy((Tbeta->c1+2),10,(Tbeta->c2+30),10,NULL);
    return;
#undef Talpha
#undef Tbeta
}
char Xalpha[428];
char Xbeta[200];
```

This translation is very similar to the one above except that the definition of the symbol `Tbeta` is accomplished via a define as opposed to a variable.

The translation below, which uses the Gs setting, is our favorite if you and your compiler can accept it.

```
void demo()
{
extern struct {
    float a[10][10],b[5],c,d;
} Xalpha;
extern struct {
    char c1[150],c2[50];
} Xbeta;
    Xalpha.a[0][0] = Xalpha.c;
    Xalpha.b[4] = Xalpha.d;
    ftnscopy((Xbeta.c1+2),10,(Xbeta.c2+30),10,NULL);
    return;
}
char Xalpha[428];
char Xbeta[200];
```

This translation comes closest to looking like the original FORTRAN, and produces relatively efficient code. In addition, there is no need to introduce extra symbols into the code. The problem is that this is technically not valid code. At best it would flunk an introductory course in C. The common blocks are being defined in different ways in different places in the code.

The translation below has the same efficiency as the one above and is perfectly valid C programming. It is produced with the Gr setting.

```
void demo()
{
extern char Xalpha[],Xbeta[];
    *(float*)Xalpha = *(float*)(Xalpha+420);
    *((float*)(Xalpha+400)+4) = *(float*)(Xalpha+424);
    ftnscopy((Xbeta+2),10,((Xbeta+150)+30),10,NULL);
    return;
}
char Xalpha[428];
char Xbeta[200];
```

Here the problem is that the COMMON storage has been stripped of all its variable names; thus, the code is difficult to read. What is `beta+150` for example. Use this flag if you are not interested in the intermediate C output or if you require special COMMON block layouts.

The simplest translation of this example is produced by the Gv setting and is shown below.

```
void demo()
{
extern float Xa[10][10],Xb[],Xc,Xd;
extern char Xc1[],Xc2[];
    Xa[0][0] = Xc;
    Xb[4] = Xd;
    ftnscopy((Xc1+2),10,(Xc2+30),10,NULL);
    return;
}
float Xa[100],Xb[5],Xc,Xd;
char Xc1[150],Xc2[50];
```

Here the COMMON variables themselves are the external symbols. If the COMMON was done carefully in the FORTRAN or if you are intending to build a library and wish to simplify the externals structure, then this setting gives a very clean and efficient translation. We use it often, but be careful.

The final translation of this example is produced by the Ga setting and is shown below.

```
void demo()
{
extern struct Xalpha {
    float a[10][10],b[5],c,d;
} *Calpha;
#define Talpha (*Calpha)
extern struct Xbeta {
    char c1[150],c2[50];
} *Cbeta;
#define Tbeta (*Cbeta)
    Talpha.a[0][0] = Talpha.c;
    Talpha.b[4] = Talpha.d;
    ftnscopy((Tbeta.c1+2),10,(Tbeta.c2+30),10,NULL);
    return;
#undef Talpha
#undef Tbeta
}
```

Note that we now have a simple pointer to a structure and no actual structures allocated. This technique is most typically used when C code is already present in the application, or when special techniques are being used to reference the COMMON blocks.

A final comment on the treatment of common blocks has to do with the need to form multiple symbols from the original COMMON identifier. By default, PROMULA.FORTRAN uses 'T', 'C', and 'X' prefixes. These conventions can be easily changed via a configuration file as discussed in the following chapter. Remember in establishing these conventions, however, that many versions of FORTRAN allow local symbols or even COMMON members to have the same identifier as a COMMON block.

2.14.1 Overall Alignment Control with Gp — Gpc, Gps, Gpl, Gpd

In examining the translation of the sample code under the Gp option, note that the common structures are defined as simple arrays of `char` with the appropriate length at the end of the compilation:

```
char Xalpha[428];
char Xbeta[200];
```

These declarations are needed to force the linker to allocate sufficient space for each common area. Now most, but not all, linkers will allocate such areas on an appropriate byte boundary to ensure that no alignment errors will occur relative to the first byte. For those linkers which do not perform this allocation automatically, the modified Gp switch can be used. The Gpc switch is the default for Gp which says to declare the external areas as simple arrays of char. Gps declares them as arrays of short, thus forcing 2-byte alignment; Gpl declares them as arrays of long, thus forcing 4-byte alignment; and Gpd declares them as arrays of double, thus forcing 8-byte alignment.

2.15 Iname — Name of File Containing Inline Functions

Various C compilers have conventions which allow you to mark certain functions as having special properties. A typical such marker is `Inline` which, when it precedes a function declaration, indicates that the code for that function is to be compiled inline. There are other such markers as well in other C environments — especially the Macintosh and Windows.

An inline functions file simply contains a list of function names, each entered on a separate line. Those functions are assigned the special marker. Thus, as an example, consider the following inline function file `test.inl`:

```
ialpha
```

when used via the command line switch `Itest` to process the following FORTRAN program:

```
PROGRAM TEST
I = IALPHA(J)
K = JALPHA(J)
STOP
END
FUNCTION IALPHA(J)
IALPHA = J * 6
RETURN
END
FUNCTION JALPHA(J)
IALPHA = J + 6
RETURN
END
```

The translation produced is as follows:

```
void main(argc,argv)
int argc;
char* argv[];
```

```
{
_Inline extern long ialpha();
extern long jalpha();
static long i,j,k;
    ftnini(argc,argv,NULL);
    i = ialpha(&j);
    k = jalpha(&j);
    STOP(NULL);
}
_Inline long ialpha(j)
long *j;
{
static long ialpha;
    ialpha = *j*6L;
    return ialpha;
}
long jalpha(j)
long *j;
{
static long jalpha,ialpha;
    ialpha = *j+6L;
    return jalpha;
}
```

Note that the function `IALPHA` has been assigned the special marker, while the function `JALPHA` has not.

The actual marker `_Inline` can be easily changed via a configuration file as discussed in the following chapter.

2.16 Target C Int Type — Is, Il

An interesting and excellent feature of C is the way in which it defines fixed point integer values as:

`short` meaning a relatively narrow range of values but requiring less storage

`long` meaning a relatively wide range of values but requiring more storage

`int` meaning the most efficient fixed point storage technique which might be short or long, depending upon the machine.

FORTRAN has no real equivalent of the `int` type. Its integers are either short or long. In PROMULA.FORTRAN, therefore, integer declarations are translated as either `short` or `long`, with `int` being used instead of the appropriate one. The I command switch tells PROMULA.FORTRAN whether `int` is short or long. Its settings are mutually exclusive and are as follows:

`Is` Specifies that `int` is short.

`Il` Specifies that `int` is long.

The default setting for this switch depends upon the environment in which PROMULA.FORTRAN is implemented. Environments with short ints have Is as the default; while environments with long ints have Il as a default. This switch should be changed ONLY if cross-compilation is intended. Specifying the wrong default int type will produce incorrect results.

The following FORTRAN code contains three types of INTEGER declarations.

```
SUBROUTINE DEMO
INTEGER*2 I,J
INTEGER*4 M,N
```

```
      INTEGER K,L
      WRITE(*,*) I,J,K,L,M,N
      RETURN
      END
```

The C code below shows the translation which has an Is setting.

```
void demo()
{
static int i,j;
static long m,n,k,l;
    WRITE(OUTPUT,LISTIO,INT2,i,INT2,j,INT4,k,INT4,l,INT4,m,INT4,n,0);
    return;
}
```

In the translation the INTEGER*2 variables are shown as `int` since `int` is `short`. The Is setting has no effect on the meaning of `INTEGER` as opposed to `INTEGER*2` versus `INTEGER*4`. The binary type of `INTEGER` is defined via the FIs and FIl switches.

The following shows the same FORTRAN translated with the Il setting.

```
void demo()
{
static short i,j;
static int m,n,k,l;
    WRITE(OUTPUT,LISTIO,INT2,i,INT2,j,INT4,k,INT4,l,INT4,m,INT4,n,0);
    return;
}
```

Note `I` and `J` are `short` and the rest are `int`. Everything else is the same. What has changed is only how the variables are labeled.

2.17 Treatment of Internally Generated Constants — Ka, Ks

If PROMULA FORTRAN encounters a constant subprogram argument which must be passed by address, it introduces a variable to contain this value. The K switch allows you to control the storage type of these variables. The K switch has two settings

Setting	Meaning
Ks	Declare internal constants as static (C bias default)
Ka	Declare internal constants as auto (FORTRAN bias default)

Consider the following simple FORTRAN fragment:

```
      SUBPROGRAM DEMO
      CALL ALPHA(I,3.0)
      STOP
      END
```

Its default translation is as follows:

```
void demo()
{
extern void alpha();
static float K1 = 3.0;
static long i;
```

```
    ftnini(argc,argv,NULL);
    alpha(&i,&K1);
    STOP(NULL);
}
```

In this translation a static variable `K1` is introduced whose value contains the appropriate constant. The address of this variable is then passed to the subprogram. The `Ks` switch produces the above. Under `Ka` the translation is as follows:

```
void demo()
{
extern void alpha();
auto float K1 = 3.0;
static long i;
    ftnini(argc,argv,NULL);
    alpha(&i,&K1);
    STOP(NULL);
}
```

In this translation the variable `K1` is declared as being auto. Under this declaration its value will be refreshed upon each invocation of the subroutine. In some cases the use of `Ka` and `Ks` can produce different results as the following discussion explains.

A constant is a fixed value. Note that in most implementations of FORTRAN, including this one, the value of a constant can be changed by passing it to a subprogram as a parameter. As an example, the following code will produce a sequence of values from 2 to 11.

```
      PROGRAM DEMO
      DO 10 I = 2,10
      CALL ALPHA(1)
      CALL BETA(1)
   10 CONTINUE
      STOP
      END
      SUBROUTINE ALPHA(I)
      I = I + 1
      RETURN
      END
      SUBROUTINE BETA(I)
      WRITE(*,*) I
      RETURN
      END
```

The default translation of this example is as follows:

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void alpha(),beta();
static long K1 = 1;
static long i;
    ftnini(argc,argv,NULL);
    for(i=2L; i<=10L; i++) {
        alpha(&K1);
        beta(&K1);
    }
    STOP(NULL);
}
void alpha(i)
long *i;
{
```

```
    *i = *i+1L;
    return;
}
void beta(i)
long *i;
{
    WRITE(OUTPUT,LISTIO,INT4,*i,0);
    return;
}
```

The reason for the sequence of values is that the value of the constant is stored in a constants table or, in the C translation, in a static variable. Multiple occurrences of the same constant use the same constant storage point; thus, the constant ends up behaving just like a variable. There are many ways around this problem, assigning the value to the variable `K1` for example; however, most FORTRAN compilers do the equivalent of the above.

Though, presumably, no one would write code such as the preceding deliberately, such code sequences do occur. This is one of those gray areas of FORTRAN that can cause problems that appear when any sort of migration is attempted. Programs that seem to work great under one FORTRAN implementation suddenly malfunction under another.

2.18 Maximum Output Line Width — Lnum

A minor issue has to do with how long to make the C output when very long statements need to be written. The default setting is 80 characters wide — the width of a typical editor window. If desired you may select any value. This flag effects the look of the output only.

A setting of zero specifies that all output associated with a given executable source statement should be placed on a single line. This setting is used with the Ln switch which places line-number directives in the C output listing. Note that some C compilers cannot process very long input lines. For these use L`max`, where `max` is the longest line width which can be accepted.

2.19 Link Time Processing of COMMON Data Modules — Lm, Ls

There is a real problem in processing C via FORTRAN having to do with the initialization of COMMON variables via DATA statements. The FORTRAN standard clearly states that COMMON variables may only be initialized in BLOCK DATA subprograms; however, most FORTRAN dialects allow usages such as the following:

```
SUBROUTINE TEST
COMMON/BLOCK/I,J
DATA I,J/5,6/
PRINT *,I,J
STOP
END
```

in which the variables `I` and `J` in the COMMON block `BLOCK` are initialized via a DATA statement.

The default translation for this subroutine looks as follows.

```
void test()
{
extern char Xblock[];
typedef struct {
    long i,j;
} Cblock;
auto Cblock *Tblock = (Cblock*) Xblock;
static int FIRST = 1;
static namelist DATAVAR[] = {
"i",Xblock,5,NULL,"j",(Xblock+4),5,NULL
};
```

```
static char *DATAVAL[] = {
"$DATAVAR",
"i=5,j=6,",
"$END"
};
    if(FIRST) {
        FIRST=0;
        fiointu((char*)DATAVAL,0,2);
        fiornl(DATAVAR,2,NULL);
    }
    WRITE(OUTPUT,LISTIO,INT4,Tblock->i,INT4,Tblock->j,0);
    STOP(NULL);
}
```

The details of this translation approach which uses an extension of the FORTRAN NAMELIST capability are discussed in the section on the Dc and Dr command line switches. In essence, the effect of this approach is that the initialization values will be assigned to the variables `I` and `J` at the time that the function `TEST` is first called.

The problem addressed by the Lm and Ls switches is that initializing these two variables at the time that `TEST` is first called may be incorrect. Many of the dialects that allow local data initializations to COMMON variables assume that those initializations are performed at link time; or, at least, that they are performed prior to the execution of any of the user's code. For these implementations `TEST` need not even be explicitly called to obtain the `I` and `J` values.

Using the Ls switch produces the following translation for the above.

```
void test()
{
extern char Xblock[];
typedef struct {
    long i,j;
} Cblock;
auto Cblock *Tblock = (Cblock*) Xblock;
static int FIRST = 1;
static namelist DATAVAR[] = {
"i",Xblock,5,NULL,"j",(Xblock+4),5,NULL
};
static char *DATAVAL[] = {
"$DATAVAR",
"i=5,j=6,",
"$END"
};
    if(FIRST) {
        FIRST=0;
        fiointu((char*)DATAVAL,0,2);
        fiornl(DATAVAR,2,NULL);
        return;                                 <==== (1)
    }
    WRITE(OUTPUT,LISTIO,INT4,Tblock->i,INT4,Tblock->j,0);
    STOP(NULL);
}
void ftnblkd() {                                <==== (2)
test();
}
```

There are two important changes in this translation relative to the previous one. Note first that a return has been added to the block of code that controls the initializations of the variables. This return allows us to perform an additional call to this function, independently of any calls that may be made by the application itself. Note second that a function called `ftnblkd` has been added to the bottom of the code which performs the initial call to `test`.

The `ftnblkd` function is the default BLOCK DATA subprogram which is called each time a program is started. Consequently, the COMMON initializations within `test` are now treated as an extension of the BLOCK DATA system and behave in the same way as BLOCK DATA initializations.

Unfortunately, this approach only works if all COMMON initializations are performed in the same module. In cases where multiple modules are being combined, perhaps via libraries, to form a single executable, there will be multiple `ftnblkd` functions created. This multiple creation will then cause a linker error.
The Om switch is used to deal with this problem. The following translation is produced for the above using Om.

```
void test()
{
extern char Xblock[];
typedef struct {
    long i,j;
} Cblock;
auto Cblock *Tblock = (Cblock*) Xblock;
static int FIRST = 1;
static namelist DATAVAR[] = {
"i",Xblock,5,NULL,"j",(Xblock+4),5,NULL
};
static char *DATAVAL[] = {
"$DATAVAR",
"i=5,j=6,",
"$END"
};
    if(FIRST) {
        FIRST=0;
        fiointu((char*)DATAVAL,0,2);
        fiornl(DATAVAR,2,NULL);
        return;
    }
    WRITE(OUTPUT,LISTIO,INT4,Tblock->i,INT4,Tblock->j,0);
    STOP(NULL);
}
void BD_test() {     <==== (3)
test();
}
```

In addition, the following message is set to standard output.

```
Creating COMMON data module: BD_test
```

With this approach, the initialization call within in each module is given a unique name, and the message alerts the user that he must provide an `ftnblkd` function which calls the individual module initialization control functions. Note that the actual convention used to form the initialization names can be easily changed via a configuration file as discussed in the following chapter.

2.20 Inclusion of Line Numbers for Debugging — Ln, L0

For platforms that support the UNIX dbx debugger or its equivalent, FORTRAN codes can be debugged using the original source FORTRAN lines. To activate this capability, the Ln switch is used. This switch tells PROMULA.FORTRAN to include line number and source file information in its intermediate C output so that it can later be used by the debugger. For best effects the following three switches should also be used: Bo, L0, Qe32000. See the relevant sections in this chapter for detailed descriptions of these switches. Bo selects the optimized bias. This is needed to give PROMULA.FORTRAN maximum flexibility in maintaining a clear correspondence between the source FORTRAN and the target C. L0 specifies that no newlines should be inserted in the listing. This is needed to maintain the line numbering system. Qe32000 allocates room for the emissions table which is needed to generate the line number information.

2.21 FORTRAN Dialect Selection Flags — Mdialect

The PROMULA.FORTRAN processor is a general-purpose, multi-dialect, and portable FORTRAN processor. It runs on multiple platforms and supports both the ANSI FORTRAN 66 and ANSI FORTRAN 77 standard dialects, as well as a large number of common extensions such as those found in the following commercial compilers: VAX FORTRAN, PDP FORTRAN, PRIME FORTRAN, Data General FORTRAN, and Sun FORTRAN. Some Fortran 90 extensions are also supported. In cases where different versions of FORTRAN have conflicting features or conventions, a dialect selection option switch can be used to select the desired set. The particular dialect options which the compiler supports are as follows:

Option	Dialect	Reference
Mvax	Vax FORTRAN	*Programming in VAX FORTRAN*, AA-D034D-TE, September 1984, Digital Equipment Corporation.
Mpdp	PDP FORTRAN	*PDP-11 FORTRAN Language Reference Manual*, AA-1855D-TC, December 1979, Digital Equipment Corporation.
Mp77	Prime FORTRAN 77	*FORTRAN 77 Reference Guide*, Fifth edition, Release T1.0-21.0, January 1988, Prime Computer, Inc.
Mpiv	Prime FORTRAN IV	*The FORTRAN Reference Guide*, FDR3057, by Anthony Lewis, March 1980, Prime Computer Inc.
Msun	Sun FORTRAN	*Sun FORTRAN Reference Manual*, 800-3418-10, March 1990, Dun Microsystems, Inc.
Mufn	Unisys FieldData FORTRAN	*Sperry Univac Series 1100 FORTRAN V Level 4R1 Programmer Reference*, 1979, Sperry Rand Corporation.
Mhny	*Honeywell FORTRAN	*RCS FORTRAN Reference Manual*, Order Number DG75, Rev. 3 (GTEDS), December 1981, GTE Data Services Incorporated.
Mf90	*Fortran 90	*Fortran 90*, X3J3/S8.115, June 1990, The FORTRAN Technical Committee of the American National Standards Institute.
Mdge	*Data General FORTRAN	*FORTRAN 77 Reference Manual*, 093-000162-02, October 1983, Data General Corporation.
Mfn5	*Cyber FORTRAN 5	*FORTRAN Version 5 Reference Manual*, July 1983, Control Data Corporation.

* Indicates partial support only

This particular option should only be used if code is being moved directly from one of these compilers to the PROMULA compiler.

2.22 Nesting Indentation to be Used in the Output — N*, N0, Nn

The N*, N0, and Nn switches specify how nesting is to be indicated in the C output. The N* switch specifies that no nesting indentation is to be used in the output. This is the default setting for the optimized bias. The N0 switch specifies that a tab character is to be used for each nesting level. The advantage of this selection is that it minimizes the size of the output file, while still showing nesting. The disadvantage is that tab characters are generally expanded to 8 characters, which makes listings containing multiple nesting levels difficult to read. The Nn switch specifies the number of blanks to be used for each output nesting level explicitly. The default indentation setting for the FORTRAN and C biases is N4. This gives a clean listing, but makes the output file somewhat larger than would be achieved via N0.

2.23 Inline Comments Output Margin Width — NCnum

The NCn switch controls the margin to be associated with inline comments when the CM1 or CM2 switches are active. The default value for this switch is 35. This value specifies that the inline comment associated with a particular output statement should be displayed on the same line as the statement beginning at column position 35. If the statement is longer than 35 characters, the comment should begin immediately after the end of the statement. Other settings have equivalent meanings. A switch setting of NC0 will force all inline comments to follow their associated statements directly.

2.24 Upper and Lower Braces Convention in C — NU0, NU1, NU2, NL1, NL2

A major area of disagreement about the readability of C has to do with the placement of braces in nested structures. If the brace placement convention does not match your desired convention, you will probably have trouble feeling comfortable with the output. By using the brace placement flags in conjunction with the indentation width switches, discussed in the sections on Nnumb and NCnum, any reasonable placement convention can be achieved.

The NU0 switch specifies that the upper brace is to be on the same line with the statement causing the nesting increase.

The NU1 switch specifies that the upper brace is on the next line at the indentation point used for the old nesting level.

The NU2 switch specifies that the upper brace is on the next line at the indentation point used for the new nesting level.

The NL1 switch specifies that the lower brace is on the next line at the indentation point for the previous nesting level.

The NL2 switch specifies that the lower brace is on the next line at the indentation point for the current nesting level.

The default is NU0, NL1 and generates the following:

```
conditional {
    statement 1
    statement 2
}
```

The user may also select the following styles:

```
   NU0, NL2

conditional {
     statement 1
     statement 2
     }
```

```
NU1, NL1

conditional
{
     statement 1
     statement 2
}
```

```
NU1, NL2

conditional
{
     statement 1
     statement 2
     }
```

```
  NU2, NL1

conditional
     {
     statement 1
     statement 2
}
```

```
NU2, NL2

conditional
     {
     statement 1
     statement 2
     }
```

2.25 Name of the File to Receive the C Output — Oname

Normally, PROMULA FORTRAN sends the intermediate C output produced to a file with the same name as the input with its extension changed to "c". If some other file is to receive the C intermediate output, then use the O switch to specify its name. There should be no whitespace between the "O" of the switch and the name of the file.

2.26 Splitting of Output into Separate Files — Os, Om

By default, PROMULA FORTRAN writes all of its translated output to a file with the same name as the input file, but with the extension changed to ".c". In some instances, if you are building a library or if you intend to do extensive editing of the output, it is convenient to have the C code corresponding to each C function written to a separate file with the same name as the function. The O switch gives you this capability. Its mutually exclusive settings are as follows:

Setting	Meaning
Om	Writes each function to a separate file with the same name as the function and with the extension ".C".
Os	Writes all output to the same file. This is the default.

Because of the fact that many C constructs, such as define and typedef, have module scope the Om flag can also be used to avoid a problem with the treatment of COMMON blocks as discussed in the section on the global variables convention flag. The cleanest treatment can be achieved via the Gs switch which defines COMMON blocks as static structures with varying internal composition. In particular, this technique defines each occurrence of the COMMON block as an external structure whose members are defined in the same manner as they are defined in the subprogram being translated. This technique produces readable code and generates relatively efficient code. The problem is that many compilers consider redefining an external with a detailed structure repeatedly in different functions to be a typing error. If you are using such a compiler, then the Om switch can be used to overcome it.

2.27 Miscellaneous Prototyping Control Flags — Pnumb, P+numb

The P command line switch introduces 15 flags for use in dealing with the prototypes and other output issues faced by PROMULA.FORTRAN. What brings these flags together is that all are primarily focused on the differences in semantics between FORTRAN and C. The flags and their values are as follows:

1	Include definitions of int functions when listing external function references.
2	Include ANSI prototypes with declarations of functions passed as arguments.
4	Exclude referenced function prototypes from the prototype output file.
8	Exclude defined function prototypes from the prototype output file.
16	Ignore prototypes for definitions.
32	Treat user prototypes as system functions.
64	Write PROMULA.FORTRAN style prototypes, not C type.
128	Write all function decls to header file.
256	Use ANSI C function declarations.
512	Make parameters always take explicit value type.
1024	Exclude undefs from the translation.
2048	Force variables to have explicit character type.
4096	Define equivalences via a #define.

8192 Use parameter identifiers in equivalences.

16384 Display include files separately.

Any combination of these flags can be obtained by summing the desired values and then associating that sum with the P switch. Alternatively, individual flags may be turned on by using the P+numb form of the switch.

2.27.1 P1 — Include Definitions of int Functions

The first flag forces the definitions of int functions when listing the external functions referenced within a given prototype. This flag is on by default. If a referenced function is not defined in a given module, then C assumes that it is an int; however, many C compilers give a warning when this assumption is made. Consider the following simple FORTRAN subroutine

```
SUBROUTINE TEST
J = 99
I = IFUNC(J)
R = RFUNC(J)
PRINT *,I,R
STOP
END
```

which references two external functions: `IFUNC` and `RFUNC`.

The default translation for this subroutine is as follows.

```
void test()
{
extern int ifunc();
extern float rfunc();
static int j,i;
static float r;
    j = 99;
    i = ifunc(&j);
    r = rfunc(&j);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
```

Notice that both `ifunc` and `rfunc` are explicitly declared. By default the P1 flag is on. Turning the P1 flag off produces the following.

```
void test()
{
extern float rfunc();
static int j,i;
static float r;
    j = 99;
    i = ifunc(&j);
    r = rfunc(&j);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
```

In this version, the fact that `ifunc` is an int is assumed by the compiler, therefore, no explicit declaration is given.

2.27.2 P2 — Use ANSI Prototypes for Argument Functions

The second flag associates a full ANSI C prototype with external function declarations passed as arguments when those functions contain arguments which are passed by value. This flag is needed only in very specialized cases where detailed prototype checking is needed. By default this flag is off. Consider the following simple test program.

```
      SUBROUTINE TEST(IFUNC,RFUNC)
      R = RFUNC(99)
      I = IFUNC(56.7)
      PRINT *,I,R
      STOP
      END
```

When processed using the Yv switch, which allows call-by-value arguments, the following C translation is produced.

```
void test(ifunc,rfunc)
int (*ifunc)();
float (*rfunc)();
{
static float r;
static int i;
    r = (*rfunc)(99);
    i = (*ifunc)(56.7);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
```

Notice that the types of the arguments are not specified in the declarations for `ifunc` and `rfunc`. Using the P2 flag produces the following result.

```
void test(ifunc,rfunc)
int (*ifunc)(float);
float (*rfunc)(int);
{
static float r;
static int i;
    r = (*rfunc)(99);
    i = (*ifunc)(56.7);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
```

Now the function declarations use the full ANSI C form for the function declarations.

2.27.3 P4,P8 — Exclude Referenced or Defined Prototypes

When processing a FORTRAN code, PROMULA.FORTRAN builds an internal table containing the prototypes for all subprograms encountered either by reference or by definition. If desired, these prototypes can be written to a file via the W switch, described elsewhere in this chapter. The P4 and P8 flags associated with the P switch block either those functions only referenced via the code or those functions only defined in the code.

The following FORTRAN code

```
      SUBROUTINE ALPHA(IA,GVAL)
      R = RFUNC(IA)
      I = IFUNC(GVAL)
      PRINT *,I,R
      STOP
      END
      FUNCTION RFUNC(I)
      RFUNC = I * I
      RETURN
      END
```

contains references to three subprograms — ALPHA, RFUNC, and IFUNC. ALPHA is defined only; RFUNC is both defined and referenced; and IFUNC is referenced only. Processing this code with the Wname switch will produce the following list of prototypes.

```
float rfunc(long*);
long ifunc(float*);
void alpha(long*,float*);
```

All three subprograms are included. Now using the P4 switch produces the following two only, since IFUNC, which was not defined, is excluded.

```
void alpha(long*,float*);
float rfunc(long*);
```

Alternatively, using the P8 switch produces the following list

```
float rfunc(long*);
long ifunc(float*);
```

in which ALPHA is excluded since it was not referenced.

The effect of these switches can be most easily seen via the use of the Wname switch which actually writes out internally generated prototypes; however, remember that PROMULA.FORTRAN can be asked to do extensive argument checking via other command line switches and via its warning message system. The P4 and P8 flags control the contexts in which internal prototypes are retained and therefore effect the operation of all of these functions.

2.27.4 P16 — Ignore Prototypes for Definitions

The P16 flag is used to modify the behavior of the Yp switch. The Yp switch is described later in this chapter. In essence, the Yp flag ensures that passed subprogram argument types are consistent. As a result of this flag, prototypes are formed when subprograms are first referenced or defined. When the first reference to a subprogram disagrees with the later definition of that subprogram, the Yp switch must enforce the prototype. This can be seen in the following FORTRAN code

```
SUBROUTINE ALPHA(IA,GVAL)
R = RFUNC(IA)
I = IFUNC(GVAL)
PRINT *,I,R
STOP
END
FUNCTION RFUNC(A)
RFUNC = ABS(A)
RETURN
END
```

in which the function RFUNC is called with an INTEGER argument; however, when the RFUNC function is later defined, that argument is a FLOAT. The translation of this code under the Yp switch is shown below.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
extern long ifunc();
extern float rfunc();
static float r;
static long i;
    r = rfunc(ia);
    i = ifunc(gval);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
```

```
float rfunc(a)
long *a;                <==========
{
static float rfunc;
    rfunc = fifiabs(*a); <=========
    return rfunc;
}
```

Note that the argument a is shown and is used as a long to agree with the earlier reference. If this behavior is not desired, the P16 flag will block the Yp switch from changing defined function argument types. The translation of the test code using both Yp and P16 becomes the following.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
extern long ifunc();
extern float rfunc();
static float r;
static long i;
    r = rfunc(ia);
    i = ifunc(gval);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
float rfunc(a)
float *a;                <=====
{
static float rfunc;
    rfunc = fifabs(*a);  <=====
    return rfunc;
}
```

2.27.5 P32 — Treat User Prototypes as System Functions

When PROMULA.FORTRAN translates a reference to an external subprogram which is not a system function, it always includes an extern declaration for that function. This is true even when user prototypes are being read by PROMULA.FORTRAN. Consider the following FORTRAN code

```
      SUBROUTINE ALPHA(IA,GVAL)
      R = RFUNC(IA)
      I = IFUNC(GVAL)
      PRINT *,I,R
      STOP
      END
      FUNCTION RFUNC(I)
      RFUNC = ABS(I)
      RETURN
      END
```

when translated using the following prototype file

```
float rfunc(long);
```

produces the following C translation.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
extern long ifunc();
```

```
extern float rfunc();  <====
static float r;
static long i;
    r = rfunc(*ia);
    i = ifunc(gval);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
float rfunc(i)
long i;
{
static float rfunc;
    rfunc = fifiabs(i);
    return rfunc;
}
```

Note that the function `rfunc` is explicitly given an extern declaration. Many users of PROMULA.FORTRAN use the same prototype files when they process FORTRAN codes and later when they compile the C outputs. In these cases the simple extern declaration of `rfunc` can interfere with the full ANSI declaration. The P32 flag simply removes the extern declarations for functions defined via user supplied prototypes. The same translation as above, with the additional use of P32, produces the following translation.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
extern long ifunc();   <====
static float r;
static long i;
    r = rfunc(*ia);
    i = ifunc(gval);
    WRITE(OUTPUT,LISTIO,INT4,i,REAL4,r,0);
    STOP(NULL);
}
float rfunc(i)
long i;
{
static float rfunc;
    rfunc = fifiabs(i);
    return rfunc;
}
```

In this translation note that `ifunc` still has an extern declaration, but that `rfunc` no longer has.

2.27.6 P64 — Write PFC Style Prototypes, not C Type

As is discussed extensively in the section on the treatment of CHARACTER variables, PROMULA.FORTRAN must add a character length argument to the subprograms which use FORTRAN CHARACTER variables. As a result, the argument list for such functions differs in their C form from their original FORTRAN form.

Consider the following FORTRAN subroutine with two CHARACTER*10 arguments.

```
SUBROUTINE ALPHA(IA,GVAL)
CHARACTER*10 IA,GVAL
PRINT *,IA,GVAL
STOP
END
```

The default prototype written for this function under the default CHd character treatment switch is as follows.

```
void alpha(char*,char*,int,int);
```

Alternatively, this prototype becomes

```
void alpha(char*,int,char*,int);
```

when the CHs switch is used. Depending upon the context, either of these prototypes would be appropriate for use with a later C compilation; however, if the intent is to use the prototypes with PROMULA.FORTRAN, then the following is needed.

```
void alpha(string,string);
```

This prototype does not have the character treatment assumption already decomposed. This prototype is produced when the P64 switch is used.

2.27.7 P128 — Write All Function decls to Header File

The following sample FORTRAN code shows a potentially serious problem for PROMULA.FORTRAN.

```
PROGRAM DEMO
CALL IALPHA("Hello"," World")
STOP
END
FUNCTION IALPHA(IA,GVAL)
CHARACTER*(*) IA,GVAL
PRINT *,IA,GVAL
IALPHA = LEN(IA) + LEN(GVAL)
END
```

This code shows a function which writes the concatenation of two strings and then returns the length of that concatenation. The only reference to this function is via a CALL statement — i.e., the fact that `IALPHA` also returns a value is being ignored.

The default translation for this code is shown below.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void ialpha();                <======
    ftnini(argc,argv,NULL);
    ialpha("Hello"," World",5,6);
    STOP(NULL);
}
long ialpha(ia,gval,P1,P2)           <======
char *ia,*gval;
int P1,P2;
{
static long ialpha;
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
    ialpha = P1+P2;
    return ialpha;
}
```

This C code will generate a fatal error from most contemporary C compilers since the status of `ialpha` is first as a void function and later as a long function. To avoid this problem, PROMULA.FORTRAN must be told to delay the writing of any function declarations until the entire code has been read. Once the best information about each function is acquired, all prototypes can be written to a file which can then be included.

The P128 flag provides this service. The translation of the above code using the P128 flag is shown below.

```
#include "test.h"          <======
void main(argc,argv)
int argc;
char* argv[];
{                          <=======
    ftnini(argc,argv,NULL);
    ialpha("Hello"," World",5,6);
    STOP(NULL);
}
long ialpha(ia,gval,P1,P2)
char *ia,*gval;
int P1,P2;
{
static long ialpha;
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
    ialpha = P1+P2;
    return ialpha;
}
```

Note that an include has been added to the front of the code and that the code itself contains no extern declarations. The file `test.h` is as follows.

```
extern long ialpha();
```

This declaration of `ialpha` is correct and reflects the best information available.

2.27.8 P256 — Use ANSI C Function Declarations

By default PROMULA.FORTRAN is designed to produce a C output which will compile on as many different platforms as possible. Consequently, it uses the old-style function declaration form which is compatible with all C compilers rather than the ANSI form which is compatible only with contemporary compilers. The default translation for the following FORTRAN code

```
      SUBROUTINE DEMO
      CALL IALPHA("Hello"," World")
      STOP
      END
      SUBROUTINE IALPHA(IA,GVAL)
      CHARACTER*(*) IA,GVAL
      PRINT *,IA,GVAL
      END
```

is as follows.

```
void demo()                 <======
{
extern void ialpha();
    ialpha("Hello"," World",5,6);
    STOP(NULL);
}
void ialpha(ia,gval,P1,P2)  <=======
char *ia,*gval;
int P1,P2;
{
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
}
```

Note that the function arguments are listed and then typed separately. The P256 flag will produce the following translation.

```
void demo(void)
{
extern void ialpha();
    ialpha("Hello"," World",5,6);
    STOP(NULL);
}
void ialpha(char *ia,char *gval,int P1,int P2)
{
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
}
```

In this form, the argument list also contains the typing information.

2.27.9 P512 — Make Parameters Always Take Explicit Value Type

There is a difference between various dialects of FORTRAN as to the treatment of untyped PARAMETERs. All contemporary dialects require that PARAMETERs be typed in the same manner as other variable symbols; however, some assign untyped PARAMETERs the type of the constant which they represent. Consider the following FORTRAN code.

```
SUBROUTINE ALPHA(IA,GVAL)
PARAMETER(AVAL = 99)
PARAMETER(IVAL = 10.2)
IA = AVAL
GVAL = IVAL
RETURN
END
```

Most compilers type `AVAL` as a float constant and `IVAL` as an integer constant. Adjustments are made to the constants themselves to obtain the correct type. To correspond to this practice, the PROMULA.FORTRAN translation of the above is as follows.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
#define aval 99.0           <====
#define ival (long)10.2     <====
    *ia = aval;
    *gval = ival;
    return;
#undef aval
#undef ival
}
```

Note that the constants themselves have changed type to accommodate the typing of the PARAMETER symbols.

The P512 flag gives the opposite interpretation, as can be seen in the following translation.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
#define aval 99         <====
#define ival 10.2       <====
    *ia = aval;
    *gval = ival;
    return;
#undef aval
#undef ival
}
```

Under this translation the constants retain their original types and the PARAMETERs are assigned the types of those constants.

2.27.10 P1024 — Exclude undefs From the Translation

Examining the translations of the preceding part brings another fundamental difference between FORTRAN and C to light. In FORTRAN, PARAMETERs have scope only within the subprogram in which they are defined; while in C the corresponding #defines have file scope. Consequently, whenever PARAMETERs are translated via #defines they must also be undefined via #undef at the end to simulate the FORTRAN scope. In those cases where FORTRAN subprograms correspond to files in C, the necessity for the #undef is removed.

The P1024 flag turns off the #undefs. The following is the same translation as presented above with the P1024 flag turned on.

```
void alpha(ia,gval)
long *ia;
float *gval;
{
#define aval 99.0
#define ival (long)10.2
    *ia = aval;
    *gval = ival;
    return;
}
```

2.27.11 P2048 — Force Variables to Have Explicit Character Type

Another issue having to do with typing is the mixture of CHARACTER and non-character information within non-integer variables. The following FORTRAN code represents the sort of situation that can occur.

```
PROGRAM DEMO
INTEGER IA(2),IB(2)
DATA IA/5Hhello/
DATA IB/6H World/
IC = IALPHA(IA,IB)
PRINT *,IC
STOP
END
FUNCTION IALPHA(IA,GVAL)
CHARACTER*(*) IA,GVAL
PRINT *,IA,GVAL
IALPHA = LEN(IA) + LEN(GVAL)
END
```

A subprogram which explicitly assumes character variables has been intermixed with one that hides characters in INTEGER variables. To make the situation clearer, the lengths of the character arguments are explicitly used by the subprogram — both in the free-form PRINT and in the computation of the return value.

The default translation for this program does not produce correct results.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern long ialpha();
static long ia[2];
static long ib[2];
static long ic;
```

```
static int ftnsiz[] = {1,1,2,1,1,2};
static namelist DATAVAR[] = {
"ia",ia,5,ftnsiz,"ib",ib,5,ftnsiz+3
};
static char *DATAVAL[] = {
"$DATAVAR",
"ia='hell','o',ib=' Wor','ld',",
"$END"
};
    ftnini(argc,argv,NULL);
    fiointu((char*)DATAVAL,0,2);
    fiornl(DATAVAR,2,NULL);
    ic = ialpha(ia,ib);             <=====
    WRITE(OUTPUT,LISTIO,INT4,ic,0);
    STOP(NULL);
}
long ialpha(ia,gval,P1,P2)
char *ia,*gval;
int P1,P2;
{
static long ialpha;
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
    ialpha = P1+P2;
    return ialpha;
}
```

When the IALPHA function is called, PROMULA.FORTRAN does not know that that function will expect the lengths of the character vectors. One way to solve this problem is to provide an explicit prototype, as shown below.

```
long ialpha(string,string);
```

Using this prototype to augment the translation, produces the following translation.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern long ialpha();
static long ia[2];
static long ib[2];
static long ic;
static int ftnsiz[] = {1,1,2,1,1,2};
static namelist DATAVAR[] = {
"ia",ia,5,ftnsiz,"ib",ib,5,ftnsiz+3
};
static char *DATAVAL[] = {
"$DATAVAR",
"ia='hell','o',ib=' Wor','ld',",
"$END"
};
    ftnini(argc,argv,NULL);
    fiointu((char*)DATAVAL,0,2);
    fiornl(DATAVAR,2,NULL);
    ic = ialpha(ia,ib,8,8);        <=====
    WRITE(OUTPUT,LISTIO,INT4,ic,0);
    STOP(NULL);
}
long ialpha(ia,gval,P1,P2)
char *ia,*gval;
int P1,P2;
{
static long ialpha;
```

```
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
    ialpha = P1+P2;
    return ialpha;
}
```

This translation is correct and will probably produce acceptable results. There are two disadvantages to the prototype approach. First, all the function prototypes must be known — note that these can be produced by PROMULA.FORTRAN. Once produced, the translation can then be done a second time using them. The other problem is that the variables `IA` and `IB` are still being treated as integer variables, even though it is clear from the context that they are being used to contain character values.

The P2048 flag forces variables to have an explicit character type when this appears to be obvious from the context. As is true of many other flags and switches, this one is a translation aid — it is not guaranteed to produce correct results in all cases. The translation of the above using P2048 without any additional prototype needed is the following.

```
void main(argc,argv)
int argc;
char* argv[];
{
extern long ialpha();
static char ia[8] = {
    'h','e','l','l','o',' ',' ',' '
};
static char ib[8] = {
    ' ','W','o','r','l','d',' ',' '
};
static long ic;
    ftnini(argc,argv,NULL);
    ic = ialpha(ia,ib,4,4);  <====
    WRITE(OUTPUT,LISTIO,INT4,ic,0);
    STOP(NULL);
}
long ialpha(ia,gval,P1,P2)
char *ia,*gval;
int P1,P2;
{
static long ialpha;
    WRITE(OUTPUT,LISTIO,STRG,ia,P1,STRG,gval,P2,0);
    ialpha = P1+P2;
    return ialpha;
}
```

Note that `IA` and `IB` are now treated as character variables. Their initialization is quite natural. Note, however, that the string lengths are allocated at their binary size, which is not the desired value in this context.

2.27.12 P4096 — Define Equivalences via a #define

Another area in which C and FORTRAN differ is "address aliasing". Basically, programs occasionally need to refer to the same area of memory in different ways. In C, pointers are used. In FORTRAN, EQUIVALENCE statements are used. The following FORTRAN program contains a simple example.

```
PROGRAM DEMO
INTEGER IA(10),IB(2)
EQUIVALENCE (IA(5),IB(2))
READ(*,*) IA
PRINT *,IB
END
```

Within this program the array `IB` is being used to refer to the fourth and fifth members of the array `IA`. The default translation for this example in C looks as follows.

```
void main(argc,argv)
int argc;
char* argv[];
{
static long ia[10];
static long *ib = (ia+3);
    ftnini(argc,argv,NULL);
    READ(INPUT,LISTIO,DO,10,INT4,ia,0);
    WRITE(OUTPUT,LISTIO,DO,2,INT4,ib,0);
}
```

In this translation the variable `IB` is defined as a pointer which points to the fourth member of the array `IA` (note that C uses zero based subscripts). This translation is relatively clean; however, it does introduce an additional pointer variable. This translation is selected as the default because the symbol `IB` remains defined, thus making debugging simpler.

An alternative translation can be produced via the P4096 flag. This translation is shown below.

```
void main(argc,argv)
int argc;
char* argv[];
{
static long ia[10];
#define ib ((ia+3))
    ftnini(argc,argv,NULL);
    READ(INPUT,LISTIO,DO,10,INT4,ia,0);
    WRITE(OUTPUT,LISTIO,DO,2,INT4,ib,0);
#undef ib
}
```

Notice that `IB` is now implemented with a define. It is not an actual memory location in the final code. Alternatively, repeated uses of `IB` might generate inefficient code if its value must be computed upon each use.

2.27.13 P8192 — Use Parameter Identifiers in Equivalences

As discussed in the section on the P4096 flag, the processing of FORTRAN EQUIVALENCE into C is difficult and clumsy at best. To complicate the situation, PARAMETER values are often used within EQUIVALENCE declarations. The following is the equivalent of the code presented above, using PARAMETERs.

```
      PROGRAM DEMO
      PARAMETER(IAOFF = 5)
      INTEGER IA(10),IB(2)
      EQUIVALENCE (IA(IAOFF),IB(IAOFF-3))
      READ(*,*) IA
      PRINT *,IB
      END
```

In the default translation, to ensure that the positions are correctly computed, the values of the parameters are used in the C produced. This can be seen in the following default translation of the above.

```
void main(argc,argv)
int argc;
char* argv[];
{
#define iaoff 5
static int ia[10];
static int *ib = (ia+3);         <=====
    ftnini(argc,argv,NULL);
    READ(INPUT,LISTIO,DO,10,INT4,ia,0);
    WRITE(OUTPUT,LISTIO,DO,2,INT4,ib,0);
#undef iaoff
```

```
}
```

The pointer expression for the IB variable is the same as when constants were used. In most cases the actual parameters forms can also be used. The P8192 flag forces this use. Using P8192, the following is produced.

```
void main(argc,argv)
int argc;
char* argv[];
{
#define iaoff 5
static int ia[10];
static int *ib = (ia+iaoff-2);   <====
    ftnini(argc,argv,NULL);
    READ(INPUT,LISTIO,DO,10,INT4,ia,0);
    WRITE(OUTPUT,LISTIO,DO,2,INT4,ib,0);
#undef iaoff
}
```

This form now uses the #defined value for IAOFF; however, a different computation must be used.

2.27.14 P16384 — Display Include Files Separately

Another issue which must be dealt with in translation has to do with include files. Ideally, FORTRAN include files should be translated into C include files. The problems involved here are two-fold. First, FORTRAN and C have very different scope rules. In FORTRAN include files are typically included within subprograms, whereas in C they are included at the front of files. The second problem is that it is often necessary to reorder the symbols when moving from FORTRAN to C. This reordering is greatly complicated by the need to retain the content of each include separately. Consider the following FORTRAN source along with two include files.

```
TEST.FOR
      PROGRAM DEMO
      INCLUDE "TEST1.INC"
      INCLUDE "TEST2.INC"
      READ(*,*) IA
      PRINT *,IB
      END
TEST1.INC
C
C     Define the needed sizes
C
      PARAMETER(NA = 100)
      PARAMETER(NB = 200)
TEST2.INC
C
C     Define the common data
C
      COMMON/ALPHA/IA(NA),IB(NB)
```

The default translation is shown below.

```
void main(argc,argv)
int argc;
char* argv[];
/*
     Define the needed sizes
*/
{
/*
     Define the common data
*/
#define na 100L
#define nb 200L
extern char Xalpha[];
```

```
typedef struct {
    long ia[na],ib[nb];
} Calpha;
auto Calpha *Talpha = (Calpha*) Xalpha;
    ftnini(argc,argv,NULL);
    READ(INPUT,LISTIO,DO,(int)(na),INT4,Talpha->ia,0);
    WRITE(OUTPUT,LISTIO,DO,(int)(nb),INT4,Talpha->ib,0);
#undef na
#undef nb
}
char Xalpha[1200];
```

In this translation the information from the include files has been incorporated into the translation.
Using the P16384 flag, three output files are created: `test.c`, `TEST1.h`, and `TEST2.h`. These are shown below.

```
test.c
    void main(argc,argv)
    int argc;
    char* argv[];
    {
    #include "TEST1.h"
    #include "TEST2.h"
        ftnini(argc,argv,NULL);
        READ(INPUT,LISTIO,DO,(int)(na),INT4,Talpha->ia,0);
        WRITE(OUTPUT,LISTIO,DO,(int)(nb),INT4,Talpha->ib,0);
    }
    char Xalpha[1200];
TEST1.h
    #ifndef ICF_TEST1
    /*
         Define the needed sizes
    */
    #define na 100L
    #define nb 200L
    #define ICF_TEST1
    #endif /*ICF_TEST1 */
TEST2.h
    #ifndef ICF_TEST2
    /*
         Define the common data
    */
    #endif /*ICF_TEST2 */
    extern char Xalpha[];
    typedef struct {
        long ia[na],ib[nb];
    } Calpha;
    auto Calpha *Talpha = (Calpha*) Xalpha;
    #ifndef ICF_TEST2
    #define ICF_TEST2
    #endif /*ICF_TEST2 */
```

The `test.c` file corresponds precisely with the original FORTRAN insofar as the placement and order of the include are concerned, since the include could well be included multiple times within a given file. Each file defines its own internal variable to keep symbols such as #defines from occurring multiple times. This convention, which is needed to overcome the scope differences between FORTRAN and C, presents an excellent opportunity to do away with the #undefs discussed earlier, at least insofar as the PARAMETERs within include files are concerned.
To give some idea of how complicated the actual production of the above files is, the listing file produced is shown below.

```
If Line# Nl Translation
-- ----- -- -----------
       1    #define SPROTOTYPE
```

```
      2    #include "fortran.h"
      3    void main(argc,argv)
      4    int argc;
      5    char* argv[];
      6    {
      7    #include "TEST1.h"
1     8    #ifndef ICF_TEST1
1     9    /*
1    10         Define the needed sizes
1    11    */
     12    #include "TEST2.h"
2    13    #ifndef ICF_TEST2
2    14    /*
2    15         Define the common data
2    16    */
1    17    #define na 100L
1    18    #define nb 200L
2    19    #endif /*ICF_TEST2 */
2    20    extern char Xalpha[];
2    21    typedef struct {
2    22        long ia[na],ib[nb];
2    23    } Calpha;
2    24    auto Calpha *Talpha = (Calpha*) Xalpha;
     25        ftnini(argc,argv,NULL);
     26        READ(INPUT,LISTIO,DO,(int)(na),INT4,Talpha->ia,0);
     27        WRITE(OUTPUT,LISTIO,DO,(int)(nb),INT4,Talpha->ib,0);
     28    }
1    29    #define ICF_TEST1
1    30    #endif /*ICF_TEST1 */
2    31    #ifndef ICF_TEST2
2    32    #define ICF_TEST2
2    33    #endif /*ICF_TEST2 */
     34    char Xalpha[1200];
```

Note that the writing of the three files is completely intertwined.

2.28 Listing File Control — PAname, PHnumb, PNname, PWnumb

As described earlier, the E flags produce various types of listing and reports about the code being translated — its content, compiled form, and its C form. The PA, PH, PN, and PW switches determine the characteristics of the file to receive the listing.

PAname specifies the name of a file to which the listing currently being produced is to be appended. If no file with the specified name exists, then one is created. PNname, on the other hand, specifies the name of a new file to receive the listing. If the named file already exists, it is truncated. If the specified name under either PA or PN has no extension, an extension of `lst` is assumed.

The PH and PW flags specify the desired output page height and width. The default setting for PH is 80 lines and that for PW is 132 characters. The reports produced are quite large, and the minimum setting allowed for PW is 80 characters. Note that the symbol table widths are adjusted to fit into the specified page size; however, the source code listing is truncated if the source line display becomes too wide.

2.29 Quantity Control Flags — QInumb, QEnumb, QDnumb, QXnumb, QHnumb, QWnumb

Generally, PROMULA examines its environment to determine the resources and values that it assigns to its internal processing controls. In some cases the user needs to be able to override the default settings for the quantities assigned, in particular, the following settings may be controlled:

Flag	Quantity controlled
QInumb	Size of compacted statement storage
QEnumb	Size of the line number table
QDnumb	Size of a data block
QXnumb	Size of external information storage
QHnumb	Size of include file information storage
QWnumb	Word size of source platform

2.29.1 QInumb — Size of Compacted Statement Storage

As PROMULA processes the raw FORTRAN source it strips all blanks and comments and stores the resulting compacted statement for compilation. The size of the storage area for this compacted statement is 5120 characters. This is sufficient for a statement with over 70 continuation lines. If extremely long statements are being processed or if memory is at a premium, the size of this area may be changed via the QInumb command line switch.

Note that if the size of the compacted statement storage area is exceeded, the fatal error 522

```
Statement contains more than nnnnn characters -- use the QIn flag to increase this
value.
```

is generated.

2.29.2 QEnumb — Size of the Line Number Table

When line number information is being generated in the C output via the Ln or DB command line switches, this switch is used to indicate the amount of storage to be allocated for this information. When Ln is used, this switch is required; when DB is used, the default value established for this switch is 32000. To change this default for DB the QEnumb switch must follow the DB switch of the command line.

Note that if the size of the line number information table is exceeded, the fatal error 532

```
The line number storage of nnnnn bytes is insufficient, use the QEn flag to increase
it.
```

is generated.

2.29.3 QDnumb — Size of a Data Block

When PROMULA encounters data initializations, it must allocate memory equal to the size of the variable to store these initializations. The values for all variables less than a data block threshold value are allocated into fixed length blocks using PROMULA's standard internal variable length record processing logic. Variables whose size exceed this threshold are allocated their own storage areas from system memory. The default setting of this threshold value is 512. On memory limited machines, large programs that do massive data initializations might want to change this value.

Since this is a tuning value, as opposed to an actual allocation value, no direct message pertaining to it is generated.

2.29.4 QXnumb — Size of External Information Storage

PROMULA saves the names and types of external symbols that have been declared via EXTERNAL statements and that have been passed to other functions, but whose final type is not known. See the discussion of the P128 prototyping control flag for detailed information on this topic. PROMULA also saves names of those subprograms that have initializations for COMMON blocks. See the discussion of link time processing of COMMON data modules — the Lm and Ls command line switches — for a discussion of this storage.

The default setting for the QXnumb switch is 2000. If the external symbols storage area is exceeded, the fatal error 519

```
The unresolved external symbol storage of nnnn bytes is insufficient, use the QXn
flag to increase it.
```

is generated.

2.29.5 QHnumb — Size of Include File Information Storage

When include files are being translated independently of the source codes containing them via the P16384 switch, a storage area 512 characters large is allocated to control this process. See the discussion of the P16384 switch for a detailed discussion of this topic. The QHnumb switch can be used to change this value.

If the include file information storage area storage area is exceeded, the fatal error 531

```
The include file storage of nnnn bytes is insufficient, use the QHn flag to increase
it.
```

is generated.

2.29.6 QWnumb — Word Size of Source Platform

The typical word size for contemporary platforms is 32 bits with 8 bits per character. This information is internally encoded as

```
(bits per word) * 100 + (bits per character)
```

Thus, the default word size specification value is 3208. When translating source codes from platforms with different word sizes, the word size value can be changed via the QWnumb switch. Note that moving codes between machines of different word sizes is potentially very difficult. The C output produced will retain the characteristics of the source platform. If explicit word size assumptions were made in the source, then those will probably not be well-formed on the target platform.

2.30 Specify a Configuration File — Rname

The R switch allows you to read additional configuration files. These files contain function prototypes and miscellaneous other configuration information. The format and use of configuration files are discussed in Chapter 3.

Suffice to say here that the Rname switch will read configuration information from the files named. The Rname switch may occur multiple times, to allow the reading of multiple files. If there is no extension supplied with the first file, then an extension of `.cnf` will be assumed. If there is no extension supplied for additional files, an extension of `.pro` is assumed. The reason for this is that, typically, when multiple files are read the first contains the configuration information; while the additional files contain function prototype information.

It should be noted that configuration files may contain command line switch information; thus, a common use of a configuration file is to supply commonly used switches independently of the command line.

2.31 Storage Threshold Values — SAnum, SDnum, SSnum, SVnum, SZnum

One of the realities that has to be faced in using FORTRAN on small machines is that FORTRAN is extremely naive in its use of memory. Though the FORTRAN 77 standards committee gave the new language a SAVE statement which was to have helped, this statement is rarely implemented. In essence, all variables in a FORTRAN program are assigned a unique and fixed memory location. If you do not have enough memory on your machine to do this, then you are in real trouble. PROMULA takes this problem head on since, as consultants, it is the one we have encountered most often in "downsizing" mainframe codes to the PC. There are four types of memory made available to you via PROMULA.

First, there is static memory. This is the default memory for variables. It is equivalent to the normal FORTRAN memory allocation. It is wasteful in that local variables all have unique memory locations, but is simple to implement. Also, static storage has memory between calls. This means that local variables can retain their values between calls to their code. This is a trick that is often used by older FORTRAN source codes.

Second, there is auto storage. This storage is allocated to the stack each time a function is called. When the function exits, the storage is returned to the stack for use by other functions. The advantages of auto storage are that it is fast, easy to use, and economizes on total storage used. The disadvantages are that it has no memory and that it is very limited on PC platforms where program stacks are typically quite short and always less than 64K on the PC.

Third, there is dynamic storage. Dynamic storage is typically taken from that storage available on the machine which is not already allocated to the program. This is generally called "the heap". In C it is accessed via the `malloc` function and is returned via the `free` function. The advantages of dynamic storage is that it is most likely to be abundant, and economizes total storage used. The disadvantages are that it must be explicitly allocated each time a function is entered and freed each time a function is exited. PROMULA automatically inserts the code to do this when dynamic storage is used.

Fourth, there is virtual storage. Virtual storage is actually maintained in a disk file. As it is accessed it is "paged" into memory. The advantages of virtual storage are that it is practically unlimited and, like static storage, has memory. Also virtual memory is unique in the fact that it remains viable after the program has completed execution. This topic is discussed in the chapter on the PROMULA interface in the FORTRAN Compiler User's Manual. The disadvantages of virtual memory are that it is slower than the other memory types and that the code using it tends to be difficult to read. Also, there are problems associated with utility functions which use both virtual and non-virtual pointer arguments.

Such functions require two versions. This topic is discussed in the chapter on configuration prototypes which allow you to deal with this problem.

The S switch allows you to specify storage threshold values for five different memory types. Its settings are as follows:

SA	Specifies that no variables are to be stored as auto. This is the default setting for this storage type.
SAnum	Where num is greater than or equal to zero, specifies that any variable whose size in bytes is greater than or equal to num but less than any other threshold setting should be stored as an auto variable.
SS	Specifies that no variables are to be stored as static.
SSnum	Where num is greater than or equal to zero, specifies that any variable whose size in bytes is greater than or equal to num but less than any other threshold setting should be stored as a static variable. The default setting for this storage type is Ss0.
SD	Specifies that no variables are to be stored as dynamic. This is the default setting for this storage type.
SDnum	Where num is greater than or equal to zero, specifies that any variable whose size in bytes is greater than or equal to num but less than any other threshold setting should be stored as a dynamic variable.
SZ	Specifies that no variables are to be stored as dynamic, but taken from the virtual file manager.
SZnum	Where num is greater than or equal to zero, specifies that any variable whose size in bytes is greater than or equal to num but less than any other threshold setting should be stored as a dynamic variable but should have its values initialized via the virtual memory manager and should have its values returned to that manager when freed.
SV	Specifies that no variables are to be stored as virtual. This is the default setting for this storage type.
SVnum	Where num is greater than or equal to zero, specifies that any variable whose size in bytes is greater than or equal to num but less than any other threshold setting should be stored as a virtual variable.

The effect of these settings is to allocate memory into size ranges. Variables are then allocated to a given storage range. The example below gives a summary of the storage switch use.

```
      SUBROUTINE DEMO
      DIMENSION A(10),B(10,10),C(10,10,10),D(10,10,10,10)
      DO 10 I = 1,10
      C(I,I,I) = D(I,I,I,4)
      B(I,I) = C(I,I,1)
      A(I) = B(I,3)
   10 CONTINUE
      RETURN
      END
```

This example contains a variety of arrays having from 40 to 40,000 bytes. The following is the default translation.

```
void demo()
{
static long i;
static float a[10],b[10][10],c[10][10][10],d[10][10][10][10];
    for(i=0L; i<10L; i++) {
        c[i][i][i] = d[3][i][i][i];
        b[i][i] = c[0][i][i];
        a[i] = b[2][i];
    }
    return;
}
```

On a PC platform this function would not compile because the `d` array is too large. Compiling the same example with SD30000 produces the following result:

```
void demo()
{
static long i;
static float a[10],b[10][10],c[10][10][10];
auto float *d;
    d=(float*)ftnalloc(40000L);
    for(i=0L; i<10L; i++) {
        c[i][i][i] = *(d+i+(i+(i+30)*10L)*10L);
        b[i][i] = c[0][i][i];
        a[i] = b[2][i];
    }
    goto ELP;
ELP:
    ftnfree((char*)d);
}
```

The three variables with less than 30,000 bytes have been declared static; while the large array has been declared dynamic. To define this dynamic array, logic has been added to the beginning and exit of the function to allocate and free the storage for the `d` array. Note that PROMULA adds this logic for you.

Now assume that we want everything to be dynamic. To do this use SS to turn off static and SD0 to make everything dynamic. The following is the result.

```
void demo()
{
auto long *i;
auto float *a,*b,*c,*d;
    a=(float*)ftnalloc(40L);
    b=(float*)ftnalloc(400L);
    c=(float*)ftnalloc(4000L);
```

```
    d=(float*)ftnalloc(40000L);
    i=(long*)ftnalloc(4L);
    for(*i=0L; *i<10L; *i+=1) {
        *(c+*i+(*i+*i*10L)*10L) = *(d+*i+(*i+(*i+30)*10L)*10L);
        *(b+*i+*i*10L) = *(c+*i+*i*10L);
        *(a+*i) = *(b+*i+20);
    }
    goto ELP;
ELP:
    ftnfree((char*)i);
    ftnfree((char*)d);
    ftnfree((char*)c);
    ftnfree((char*)b);
    ftnfree((char*)a);
}
```

This probably went too far. We really did not want the simple variable `i` to be dynamic. Try again with SS to turn off static, SA0 to use auto for small variables, and SD10 to make everything else dynamic. The following is the result.

```
void demo()
{
auto long i;
auto float *a,*b,*c,*d;
    a=(float*)ftnalloc(40L);
    b=(float*)ftnalloc(400L);
    c=(float*)ftnalloc(4000L);
    d=(float*)ftnalloc(40000L);
    for(i=0L; i<10L; i++) {
        *(c+i+(i+i*10L)*10L) = *(d+i+(i+(i+30)*10L)*10L);
        *(b+i+i*10L) = *(c+i+i*10L);
        *(a+i) = *(b+i+20);
    }
    goto ELP;
ELP:
    ftnfree((char*)d);
    ftnfree((char*)c);
    ftnfree((char*)b);
    ftnfree((char*)a);
}
```

As a final example, do the same thing as above except replacing dynamic storage with virtual storage. To do this enter

```
SS SA0 SV10
```

The following is the result.

```
void demo()
{
auto long i;
static long a=32,b=72,c=472,d=4472;
    for(i=0L; i<10L; i++) {
        *(float*)vmsdel(c+(i+(i+i*10L)*10L)*4) = *(float*)vmsuse(d+(i+(i+(i+30)*
          10L)*10L)*4);
        *(float*)vmsdel(b+(i+i*10L)*4) = *(float*)vmsuse(c+(i+i*10L)*4);
        *(float*)vmsdel(a+i*4) = *(float*)vmsuse(b+(i+20)*4);
    }
    return;
}
```

Now the variables have become long static values and references to the variables have been replaced with function references.

2.32 FORTRAN Dialect Doloop Assumptions — T0, T1, T2

Below is the syntax for the FORTRAN DO statement. It looks very simple; however, what you see is unfortunately not what you get. FORTRAN compilers vary widely on how the DO statement is executed.

Syntax:

```
DO [slab[,]] v=e1,e2[,e3]
```

Where:

- `slab` is the label of an executable statement called the terminal statement of the DO loop. If slab is omitted, the DO loop is terminated via an END DO statement.
- `v` is an integer, real, or double precision control variable.
- `e1` is an initial parameter.
- `e2` is a terminal parameter.
- `e3` is an optional increment parameter; default is 1.

`e1`, `e2`, and `e3` are called indexing parameters; they can be integer, real, double precision, or symbolic constants, variables, or expressions.

The DO statement differs widely between FORTRAN 66 and FORTRAN 77. In FORTRAN 66, the value of `v` is not compared with that of `e2` until the bottom of the loop; therefore, the loop is always executed once. In FORTRAN 77, the comparison of `v` is performed at the top of the loop; therefore, if `e1` exceeds `e2` initially the loop is never incremented. Officially, do loops are to be executed as follows:

(1) The expressions `e1`, `e2`, and `e3` are evaluated and then converted to the type of the control variable `v`, if necessary, yielding the values `m1`, `m2`, and `m3`.

(2) The control variable `v` is assigned the value of `m1`.

(3) The iteration count is established as follows:

```
ic = MAX(INT((ms-m1+m3)/m3),mtc)
```

where `mtc` is the minimum iteration count and equals 0 if FORTRAN 66 conventions are desired and 1 if FORTRAN 77 conventions are desired.

(4) If `ic` is not zero, the loop is executed, else execution continues beyond the end of the loop.
(5) The control variable is incremented by the value `m3`, `ic` is decremented by 1, and execution loops back to step 4.

To translate DO loops in the official manner then requires introducing two temporary variables for each DO loop: the iteration counter and the value of the increment since this increment may be changed within the loop. Consider the following FORTRAN subroutine:

```
      SUBROUTINE DEMO
      INTEGER V,E1,E2,E3
      DO 10 V = E1, E2, E3
      WRITE(*,*) V
10    CONTINUE
      RETURN
      END
```

In this loop all of the controls are variable. The translation of this loop using the 77 conventions looks as follows:

```
void demo()
{
static int v,e1,e2,e3,D1,D2;
    for(v=e1,D1=e3,D2=(e2-v+D1)/D1; D2>0; D2--,v+=D1) {
        WRITE(OUTPUT,LISTIO,INT4,v,0);
    }
    return;
}
```

Note that there are two temporary variables introduced. `D1` contains the value of the increment and `D2` contains the value of the iteration count. Thus the first statement in the C `for` loop performs steps 1, 2, and 3 from above. The second statement in the C `for` loop performs the step 4 logic, and the last statement performs the step 5 logic. Note that the calculations are arranged in such a manner that none of the original `e1`, `e2`, or `e3` expressions need to be evaluated more than once.

PROMULA also performs two other simplifications with `for` loops. First, if the body of the loop can be reduced to a single statement, then the simple form of the `for` statement is used. Also, the statement number is deleted if it is only being used as the terminating point of the loop.

When 66 conventions are in effect, the loop must be forced to execute at least once. To implement this the C ternary operator ?: is used. The following is the translation of the above, with the T1 command line option set, which requests 66 style DO loops.

```
void demo()
{
static int v,e1,e2,e3,D1,D2;
    for(v=e1,D1=e3,D2=(D2=(e2-v+D1)/D1)>0?D2:1; D2>0; D2--,v+=D1) {
        WRITE(OUTPUT,LISTIO,INT4,v,0);
    }
    return;
}
```

Since many are not familiar with this notation the expression

```
D2 = (D2=(e2-v+D1)/D1) > 0 ? D2 : 1
```

has the following value. Using `D2` as a temporary variable, set it equal to the value of the iteration count as before. If that temporary value is greater than 0, set `D2` equal to it; else set `D2` equal to 1. Clearly this is the desired value. Note that other approaches introduce an additional temporary variable which is set true initially and false on increment which forces execution of the loop at least once. Clearly the approach here is cleaner.

Of course, the same logic as above holds for loops with floating point controls. The following is an example.

```
   SUBROUTINE DEMO
   DO 10 V = E1, E2, E3
   WRITE(*,*) V
10 CONTINUE
   RETURN
   END
```

This is the same as before, but `E1`, `E2`, `E3`, and `V` are all floating point. The translation is as follows:

```
void demo()
{
static int D2;
static float v,e1,e2,e3,D1;
    for(v=e1,D1=e3,D2=(e2-v+D1)/D1; D2>0; D2--,v+=D1) {
        WRITE(OUTPUT,LISTIO,REAL4,v,0);
    }
    return;
```

```
}
```

The result is almost identical except that `D2`, the iteration count, is typed as a fixed point value as is required by the specification. On the other hand, `D1` which contains the increment is the same type as the iteration variable.

The above general translation is needed only if one of the following is true:

(1) The increment is a variable expression

(2) The maximum is a complex expression or is changed inside the loop.

(3) The control variable is changed inside the loop.

Most DO loops do not meet the above criteria; therefore, a much simpler form of the loop can be used. Consider the following which shows the same example as before, except the increment is fixed constant.

```
      SUBROUTINE DEMO
      INTEGER V,E1,E2
      DO 10 V = E1, E2, 3
      WRITE(*,*) V
10    CONTINUE
      RETURN
      END
```

Now there is no need to introduce any temporary variables since the simplistic view of the DO loop gives the same result.

```
void demo()
{
static int v,e1,e2;
    for(v=e1; v<=e2; v+=3) {
        WRITE(OUTPUT,LISTIO,INT4,v,0);
    }
    return;
}
```

The final point to be made about DO loops is their close relation to array subscripts. In many instances the DO loop control variables can be reduced if the control variable is used only as an array subscript. The following example shows this.

```
      SUBROUTINE DEMO
      INTEGER V,E2
      DIMENSION A(10,10)
      DO 10 V = 1, E2
      A(V,V) = 10.0
10    CONTINUE
      RETURN
      END
```

In this example the source of the array subscripts are DO loop counters and those counters are used only as subscripts; therefore, the counters can be reduced, thus simplifying the subscript expression and simplifying the translation of the DO loop. This particular optimization derives from a particular pattern in FORTRAN programs. It is unlikely that any generalized optimizer found in a C compiler would detect or perform this optimization.

```
void demo()
{
static int v,e2;
static float a[10][10];
    for(v=0; v<e2; v++) {
        a[v][v] = 10.0;
    }
```

```
    return;
}
```

PROMULA supports both FORTRAN 66 and FORTRAN 77 dialects. In most cases this merely means ignoring various restrictions placed on the language in the 77 standard and then adding the 77 constructs. There is no real conflict between the two dialects except for one thing — DO statements. DO is probably the most used statement in the language, except for assignment. What was in the minds of the committee? It must have been a very long night. Anyway, with FORTRAN 66 the following code fragment will compute a value of 1; while in 77 it will produce a value of zero.

```
      IVAL = 0
      N = 0
      DO 10 I = 1,N
      IVAL = IVAL + 1
10    CONTINUE
```

This is because in 77 a DO loop is not executed if at the initial entry the minimum exceeds the maximum; while in 66 all loops are always executed once.

The above is only half of the nightmare. As we have discussed earlier, when the 77 standard was announced, vendors had to find a way to still support the older programs. Without fail, all FORTRAN compilers vended since the 77 standard have supported a control switch which tells the compiler what to do about the DO statement. As will be discussed here, T is our switch. The problem is that control switches are rarely part of the actual code; therefore, when you have in front of you a code with logic like the above in it, you have no idea what the expected result is. There is no right translation. Did the programmer do it intentionally or unintentionally? Did he know that there were two standards? If he did, then which was he using? If he did not, then what did the JCL that someone probably gave him have in it?

At any rate, if you are processing your own code, then you probably know which dialect you are using. If you are not translating your own code, then make a guess. The individual settings associated with this flag are as follows:

T0 The minimum DO statement trip count is zero as specified in the FORTRAN 77 standard. This is the default setting for this switch.

T1 The minimum DO statement trip count is one as specified in the FORTRAN 66 standard.

T2 The minimum DO statement trip count is zero as specified in the FORTRAN 77 standard. Always use the formal FORTRAN 77 trip count computation when iterating through loops.

The DO statement is, of course, translated via the for statement in C. The following is a simple translation to show the difference between T0, T1 and T2.

```
      SUBROUTINE DEMO
      IVAL = 0
      N = 0
      DO 10 I = 1,N
      IVAL = IVAL + 1
10    CONTINUE
      RETURN
      END
```

The following translation was produced using the default T0 setting.

```
void demo()
{
static int ival,n,i;
    ival = 0;
    n = 0;
    for(i=1; i<=n; i++) {
        ival = ival+1;
```

```
    }
    return;
}
```

This translation conforms with the 77 standard because C checks the conditional part of the for statement prior to the first execution. The following shows the same thing using the T1 setting.

```
void demo()
{
static int ival,n,i;
    ival = 0;
    n = 0;
    for(i=1; i==1 || i<=n; i++) {
        ival = ival+1;
    }
    return;
}
```

To achieve the desired result, an extra condition has been placed into the for statement to force it through the code at least once.

Finally, the following shows the same thing translated via the T2 switch.

```
void demo()
{
static int ival,n,i,D2;
    ival = 0;
    n = 0;
    for(i=1,D2=(n-i+1); D2>0; D2--,i+=1) {
        ival = ival+1;
    }
    return;
}
```

This switch forces the computation of the trip count variable as specified by the FORTRAN 77 standard.

2.33 Treatment of Internally Generated Temporaries — Ta, Ts

During the processing of FORTRAN codes, PROMULA must occasionally introduce temporary variables. The contexts which generate these variables are as follows:

(1) When a computed value must be passed by reference to a subprogram.

(2) When a DO statement must compute a trip count value to control the number of times it executes.

(3) When a variable DIMENSION contains a computed value.

All of these contexts are discussed elsewhere in this chapter. This section discusses the storage status of the temporary variables generated. The following is a simple code that generates all three types of temporary variables.

```
   SUBROUTINE DEMO(A,N,M,INC)
   DIMENSION A(N+1,M+1)
   INTEGER V
   DO 10 V = 1, M+1, INC
   CALL TEST(A(V,V)/10.0)
10 CONTINUE
   RETURN
   END
```

The default C intermediate output for this example looks as follows.

```
void demo(a,n,m,inc)
int *n,*m,*inc;
float *a;
{
extern void test();
static int T1,T2,v,D4,D5;
static float T3;
    T1 = *n+1;
    T2 = *m+1;
    for(v=1,D4= *inc,D5=(*m+1-v+D4)/D4; D5>0; D5--,v+=D4) {
        T3 = *(a+v-1+(v-1)*T1)/10.0;
        test(&T3);
    }
    return;
}
```

The temporaries `T1` and `T2` are introduced to contain the values of the variable dimensions for the array `a`. The temporaries `D4` and `D5` are introduced to contain the increment values and trip count value for the DO statement. Finally, the temporary `T3` is introduced to contain the temporary value whose address must be passed to subroutine `test`. Notice that all of these temporaries are assigned to static storage because the default storage for all variables is static (see the section on the storage control flags). If the above example were compiled using a different storage convention — say SS SA0 — then the following output would be produced.

```
void demo(a,n,m,inc)
int *n,*m,*inc;
float *a;
{
extern void test();
auto int T1,T2,v,D4,D5;
auto float T3;
    T1 = *n+1;
    T2 = *m+1;
    for(v=1,D4= *inc,D5=(*m+1-v+D4)/D4; D5>0; D5--,v+=D4) {
        T3 = *(a+v-1+(v-1)*T1)/10.0;
        test(&T3);
    }
    return;
}
```

As requested, all variables including the temporary ones are now assigned to auto storage. The default setting of the temporary storage flag Ts specifies that temporary variables have "standard" storage — i.e., that they be assigned a storage class in the same manner as user defined variables.

The `Ta` flag requests that all temporaries be assigned to auto storage regardless of the rules being used to allocate other variables. Using the `Ta` flag, the following output is generated.

```
void demo(a,n,m,inc)
int *n,*m,*inc;
float *a;
{
extern void test();
auto int T1,T2,D4,D5;
auto float T3;
static int v;
    T1 = *n+1;
    T2 = *m+1;
    for(v=1,D4= *inc,D5=(*m+1-v+D4)/D4; D5>0; D5--,v+=D4) {
        T3 = *(a+v-1+(v-1)*T1)/10.0;
```

```
            test(&T3);
        }
        return;
    }
```

Under this convention, the program variable v is assigned to static storage using the standard storage rules; however, the temporary variables generated internally are all assigned to auto storage.

2.34 Specifying Unit Numbers — UR, URnum, UP, UPnum, UW, UWnum

FORTRAN has a variety of contexts with I/O statements in which no explicit unit number is provided: PRINT, READ(*, WRITE(*, PUNCH, ACCEPT, etc. There are a variety of conventions as to what actual units to associate with these statement forms. The U command line switch allows you to control this number. Note that the PROMULA FORTRAN runtime library associates no significance to any particular unit number value. A unit number may be any integer value.

The URnum command line switch tells PROMULA to use unit number num with READ or ACCEPT statements for which no unit number is explicitly shown. The default setting of UR specifies that READ statements for which no unit is explicitly shown should be directed to standard input.

The UWnum command line switch tells PROMULA to use unit number num with WRITE or PRINT statements for which no unit number is explicitly shown. The default setting UW specifies the WRITE or PRINT statements for which no unit number is explicitly shown should be directed to standard output.

The UPnum command line switch tells PROMULA to use unit number num with PUNCH statements for which no unit number is explicitly shown. The default setting UP specifies that PUNCH statements for which no unit number is explicitly shown should be directed to standard output.

Note that many FORTRAN runtime systems attach special significance to certain unit number values — i.e., many assume the unit 5 is standard input; while unit 6 is standard output. Others assume that unit 0 is standard input. Especially interesting is the Prime dialect which assumes that unit 1 is both standard input or standard output, depending upon the context of its use. See the discussion in the FORTRAN compiler manual on controlling runtime behavior for a discussion of this topic.

The U command line switches establish unit numbers, they do not establish runtime unit number file connections. This is done at runtime.

2.35 File to Receive Prototype Definitions — Wname

The W switch allows you to write function prototype files. These files are used to define the arguments and types of functions being called with the FORTRAN being translated. The format of prototype files conforms to the ANSI C standard.

The Wname switch will write prototypes either for functions referenced or defined or both. Which prototypes are written are controlled by the "P" command line switch.

If there is no extension supplied with the name of the file, then an extension of .pro will be assumed.

2.36 Miscellaneous Control Flags — Y1, Y2

There are various translation options which are difficult to classify under a general topic. The Y1 and Y2 switches control these. The Y1 switch changes the treatment of entry points; while the Y2 switch controls the output form of parameter identifiers.

Of course, as with the other numeric flags, the composite switch Y3 performs both operations.

2.36.1 The Treatment of Entry Points — Y1

The following FORTRAN source code contains a main entry and two entry points. The parameters associated with the entries all differ. In addition, the logic of the construct assumes that the value of `B` set via the call to `ISET_BVALUE` will be remembered in the calls to `IDOUBLE` and to `MULTIPLY_A`.

```
CALL ISET_BVALUE(10)
WRITE(*,*) IDOUBLE(5),MULTIPLY_A(2,4)
END
INTEGER FUNCTION IDOUBLE(A)
INTEGER A,R,B
IDOUBLE = A * B
RETURN
ENTRY MULTIPLY_A(A,R)
A = A * R
IDOUBLE = A * B
ENTRY ISET_BVALUE(B)
RETURN
END
```

We recently processed a large body of VAX-FORTRAN code which made extensive use of this ability to successively set argument values though independent entry points. The default translation produces the following C output:

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void iset_bvalue();
extern long idouble(),multiply_a();
static long K1 = 10;
static long K2 = 5;
static long K3 = 2;
static long K4 = 4;
    ftnini(argc,argv,NULL);
    iset_bvalue(&K1);
    WRITE(OUTPUT,LISTIO,INT4,idouble(&K2),INT4,multiply_a(&K3,&K4),0);
}
static long E0000(IENTRY,a,r,b)
int IENTRY;
long *a,*r,*b;
{
static long idouble;
    switch(IENTRY) {
        case 0: goto IDOUBLE;
        case 1: goto MULTIPLY_A;
        case 2: goto ISET_BVALUE;
    }
IDOUBLE:
    idouble = *a * *b;
    return idouble;
MULTIPLY_A:
    *a = *a * *r;
    idouble = *a * *b;
ISET_BVALUE:
    return idouble;
}
long idouble(a)
long *a;
{
    return E0000(0,a,NULL,NULL);
}
long multiply_a(a,r)
```

```
long *a,*r;
{
    return E0000(1,a,r,NULL);
}
long iset_bvalue(b)
long *b;
{
    return E0000(2,NULL,NULL,b);
}
```

This translation treats the individual entry points as if all argument values were reinitialized at each entry point. It is correct for most FORTRANs, but is not correct for the assumptions made in the example above. Using the Y1 flag produces the following:

```
void main(argc,argv)
int argc;
char* argv[];
{
extern void iset_bvalue();
extern long idouble(),multiply_a();
static long K1 = 10;
static long K2 = 5;
static long K3 = 2;
static long K4 = 4;
    ftnini(argc,argv,NULL);
    iset_bvalue(&K1);
    WRITE(OUTPUT,LISTIO,INT4,idouble(&K2),INT4,multiply_a(&K3,&K4),0);
}
static long *E0a,*E0r,*E0b;
static long E0(IENTRY)
int IENTRY;
{
static long idouble;
    switch(IENTRY) {
        case 0: goto IDOUBLE;
        case 1: goto MULTIPLY_A;
        case 2: goto ISET_BVALUE;
    }
IDOUBLE:
    idouble = *E0a * *E0b;
    return idouble;
MULTIPLY_A:
    *E0a = *E0a * *E0r;
    idouble = *E0a * *E0b;
ISET_BVALUE:
    return idouble;
}
long idouble(a)
long *a;
{
    E0a = a;
    return E0(0);
}
long multiply_a(a,r)
long *a,*r;
{
    E0a = a;
    E0r = r;
    return E0(1);
}
long iset_bvalue(b)
long *b;
{
    E0b = b;
```

```
    return E0(2);
}
```

Using this approach, each time the main entry point uses copies of the arguments for the other entries. This approach produces the desired result for this interpretation.

2.36.2 Output Form of Parameter Identifiers — Y2

Normally, PROMULA converts all identifiers to lower case. Thus, the following code

```
SUBROUTINE DEMO(IV)
PARAMETER GoodBye = 0, Hello = 1
IF(IV .EQ. Hello) WRITE(*,*) 'Hello'
IF(IV .EQ. GoodBye) WRITE(*,*) 'GoodBye'
END
```

produces the following default translation:

```
void demo(iv)
long *iv;
{
#define goodbye 0
#define hello 1
    if(*iv == hello) WRITE(OUTPUT,LISTIO,STRG,"Hello",5,0);
    if(*iv == goodbye) WRITE(OUTPUT,LISTIO,STRG,"GoodBye",7,0);
#undef goodbye
#undef hello
}
```

Here, the parameter identifiers have been converted to lower case. It is occasionally desirable, at least for parameter values, to retain the original notation. The Y2 flag produces the following C output.

```
void demo(iv)
long *iv;
{
#define GoodBye 0
#define Hello 1
    if(*iv == Hello) WRITE(OUTPUT,LISTIO,STRG,"Hello",5,0);
    if(*iv == GoodBye) WRITE(OUTPUT,LISTIO,STRG,"GoodBye",7,0);
#undef GoodBye
#undef Hello
}
```

In this translation the parameters have the same case conventions that they had in the FORTRAN source.

2.37 Treatment of Multiple Assignments — Xa, Ya

C allows multiple assignments to the same value to be written together. When the Ya switch is active, the processor looks for such assignments and combines them whenever possible. The Xa switch excludes this optimization. The Ya flag is active for the C and optimized biases. The FORTRAN bias does not by default perform these combinations, since they destroy the correspondence between source and output statements. Therefore, for the default FORTRAN bias Xa is active.

The following code shows a FORTRAN subprogram which contains multiple assignments — i.e., the `ival` and `lval` vectors are both being set equal to 10.

```
SUBROUTINE DEMO
DIMENSION IVAL(10), JVAL(10)
DO 10 I = 1,10
IVAL(I) = 10
```

```
      JVAL(I) = 10
10 CONTINUE
      RETURN
      END
```

The default intermediate C output for this subprogram is shown below.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=0; i<10; i++) {
        ival[i] = 10;
        jval[i] = 10;
    }
    return;
}
```

The C output maintains a one-to-one correspondence with the FORTRAN original. The multiple assignments are not simplified. The same fragment under the Ya flag produces the following.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=0; i<10; i++) {
        ival[i] = jval[i] = 10;
    }
    return;
}
```

Since Ya allows multiple assignments, the two assignment statements have been simplified into one. Notice that this simplification precedes the braces removal operation. Thus, the subprogram above processed under Ya and Yb produces the following result.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=0; i<10; i++) ival[i] = jval[i] = 10;
    return;
}
```

Now the two assignments have been collapsed into a single statement and the for loop now contains only that statement. Consequently under the Yb flag the braces can be removed. The Yb flag is discussed further in another section in this chapter.

2.38 Treatment of Single Statement Nesting Brace — Xb, Yb

C allows any conditional statement to form a compound statement with a single statement, while FORTRAN allows this only for the IF statement. When the YB switch is active, this compounding is performed whenever possible; while when the XB switch is active, a compound statement is formed only if the source was compound. YB is the default for the C and optimized biases; while XB is the default for the FORTRAN bias.

The following simple subprogram contains a DO loop, a simple IF statement, and a block IF statement.

```
      SUBROUTINE DEMO
      DIMENSION IVAL(10), JVAL(10)
      DO 10 I = 1,10
      IVAL(I) = 10
      JVAL(I) = 10
10 CONTINUE
      IF(I .EQ. 10) WRITE(*,*) 'I equals 10'
```

```
      IF(I .EQ. 11) THEN
          WRITE(*,*) 'I equals 11'
      END IF
      RETURN
      END
```

The default C intermediate output for this subprogram looks as follows.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=1; i<=10; i++) {
        ival[i-1] = 10;
        jval[i-1] = 10;
    }
    if(i == 10) WRITE(OUTPUT,LISTIO,STRG,"I equals 10",11,0);
    if(i == 11) {
        WRITE(OUTPUT,LISTIO,STRG,"I equals 11",11,0);
    }
    return;
}
```

In this translation only the simple IF statement is treated as a compound statement in C. The default C output attempts to maintain a one-to-one correspondence between the FORTRAN original and the C.

Processing the above subprogram with the Yb flag produces the following C output.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=1; i<=10; i++) {
        ival[i-1] = 10;
        jval[i-1] = 10;
    }
    if(i == 10) WRITE(OUTPUT,LISTIO,STRG,"I equals 10",11,0);
    if(i == 11) WRITE(OUTPUT,LISTIO,STRG,"I equals 11",11,0);
    return;
}
```

Now the block IF in FORTRAN has also been collapsed to a compound if in C. The Yb flag looks for every opportunity to remove unneeded braces. The DO loop is not collapsed since it contains two statements. There is another switch, however, Ya which allows multiple assignments to be collapsed into a single statement. This switch is discussed fully in another section of this manual.

The C output using both Ya and Yb produces the following.

```
void demo()
{
static int ival[10],jval[10],i;
    for(i=1; i<=10; i++) ival[i-1] = jval[i-1] = 10;
    if(i == 10) WRITE(OUTPUT,LISTIO,STRG,"I equals 10",11,0);
    if(i == 11) WRITE(OUTPUT,LISTIO,STRG,"I equals 11",11,0);
    return;
}
```

Now the for loop is also reduced to a compound statement, since it now contains a single multiple assignment.

2.39 Constant Reduction Optimization — Xc, Yc

The FORTRAN input processor often generates sequences of constant integer calculations which can be reduced prior to their output in the C. The Yc switch allows this simplification to be performed and is the default for all biases. The Xc switch excludes this optimization.

As an example of this constant reduction operation, consider the following subprogram which contains various constant subscripts.

```
SUBROUTINE DEMO(A,B,C)
DIMENSION A(10),B(6,7),C(3,4,5)
A(5) = 5.0
B(2,3) = 6.0
C(1,2,3) = 7.0
RETURN
END
```

The default intermediate C form for this fragment is as follows.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
    a[4] = 5.0;
    b[2][1] = 6.0;
    c[2][1][0] = 7.0;
    return;
}
```

Notice that in this output the values of the subscripts have been reduced by one over their FORTRAN original values. This is because in FORTRAN subscripts have a default starting value of one; while in C they always start at zero. That this reduction is in fact being performed can be seen via the Xc switch which blocks constant reduction. The C output with the Xc flag active looks as follows.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
    a[5-1] = 5.0;
    b[3-1][2-1] = 6.0;
    c[3-1][2-1][1-1] = 7.0;
    return;
}
```

The subtraction by one is now explicit in the C output. This reduction process is even more obvious when pointer notation is used for array references. The Ys flag discussed in another section of this chapter generates pointer style subscript expressions. Using this flag alone produces the following.

```
void demo(a,b,c)
float *a,*b,*c;
{
    *(a+4) = 5.0;
    *(b+13) = 6.0;
    *(c+27) = 7.0;
    return;
}
```

The values of "4", "13", and "27" are now computed from fairly complex expressions which can be seen in the following — produced using Xc and Ys.

```
void demo(a,b,c)
float *a,*b,*c;
{
    *(a+5-1) = 5.0;
```

```
    *(b+2-1+(3-1)*6) = 6.0;
    *(c+1-1+(2-1+(3-1)*4)*3) = 7.0;
    return;
}
```

It is suggested that the Yc flag always be left active; however, if you want to see where some values such as those shown above are coming from, then Xc is the appropriate flag to use.

2.40 Character Optimization Switches — Xch, Ych

The FORTRAN CHAR and ICHAR functions convert between character representations and their integer values. In most cases, no actual function reference is needed — simple assignments are sufficient. The default Ych switch removes these unneeded calls to these functions. Consider the following FORTRAN code

```
SUBROUTINE DEMO(IV)
CHARACTER*1 C
I = ICHAR('8')
C = CHAR(56)
END
```

with references to both functions.

Using the default Ych switch, the following is produced.

```
void demo(iv)
long *iv;
{
static char c[1];
static long i;
    i = '8';
    *c = 56;
}
```

Alternatively, the Xch switch produces the following translation.

```
void demo(iv)
long *iv;
{
static char c[1];
static long i;
    i = fifichar("8");
    *c = *fifchar(56);
}
```

This version would only be needed if some special character conversion logic were needed.

2.41 Treatment of FORTRAN "D" debugging statements

In the default FORTRAN dialect supported, any FORTRAN statement with a non-whitespace, non-numeric character in column 1 is treated as a comment. The Yd command line switch modifies this rule slightly. When Yd is active, statements with a "D" in column 1 are treated as though the "D" were a blank — i.e., they are not comments. This facility is used for inserting debugging statements into a FORTRAN program. The default "Xd" flag treats "D" statements as simple comments.

Consider the following simple FORTRAN subprogram, which contains a "D" type comment.

```
      SUBROUTINE DEMO(A,B,C)
      DIMENSION A(10),B(6,7),C(3,4,5)
D     WRITE(*,*) 'Demo successfully entered'
```

```
      A(5) = 5.0
      B(2,3) = 6.0
      C(1,2,3) = 7.0
      RETURN
      END
```

The default intermediate C output for this subprogram looks as follows.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
/*
     WRITE(*,*) 'Demo successfully entered'
*/
    a[4] = 5.0;
    b[2][1] = 6.0;
    c[2][1][0] = 7.0;
    return;
}
```

The "D" statement is literally transformed into a comment in the C. Using the Yd flag, however, produces the following intermediate C output.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
    WRITE(OUTPUT,LISTIO,STRG,"Demo successfully entered",25,0);
    a[4] = 5.0;
    b[2][1] = 6.0;
    c[2][1][0] = 7.0;
    return;
}
```

In this treatment the statement is compiled as though it had been a normal statement.

2.41.1 Treatment of Other Debugging Statements — Ydstring

The D debugging feature, though it is very common among FORTRAN dialects, is not part of the standard; therefore, it is fair game for alternative implementations. The YSstring statement allows alternative ways of identifying debugging statements. Consider the following FORTRAN code.

```
      SUBROUTINE DEMO(A,B,C)
      DIMENSION A(10),B(6,7),C(3,4,5)
*?      WRITE(*,*) 'Demo successfully entered'
      A(5) = 5.0
      B(2,3) = 6.0
      C(1,2,3) = 7.0
      RETURN
      END
```

Note that the *? is used to shift the statement left as well commenting it out. The default translation for this code is as follows.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
/*
?      WRITE(*,*) 'Demo successfully entered'
*/
    a[4] = 5.0;
```

```
    b[2][1] = 6.0;
    c[2][1][0] = 7.0;
    return;
}
```

Using the YD*? switch produces the following translation.

```
void demo(a,b,c)
float a[],b[7][6],c[5][4][3];
{
    WRITE(OUTPUT,LISTIO,STRG,"Demo successfully entered",25,0);
    a[4] = 5.0;
    b[2][1] = 6.0;
    c[2][1][0] = 7.0;
    return;
}
```

2.42 Use of Printf-Style Formatting — Xf, Yf

C and FORTRAN have very different ways of specifying coded write conversions. For the FORTRAN bias, all of the original FORTRAN machinery is maintained. The actual WRITE statement is translated into a C WRITE statement which consists of a series of keywords followed by the parameters associated with those keywords. In the C bias, FORTRAN WRITE statements are translated into the C printf or fprintf functions whenever possible. When not possible, the C bias uses the same translation as the FORTRAN bias. The Xf switch turns the printf conversion off, while the Yf switch turns it on.

The following short code contains a formatted write statement and a list-directed write statement.

```
      SUBROUTINE DEMO
    1 FORMAT(1X,2I5,3F10.2)
      WRITE(*,1) I,J,A,B,C
      WRITE(*,*) I,J,A,B,C
      END
```

The default intermediate C output for this code is as follows.

```
void demo()
{
static int i,j;
static float a,b,c;
static char* F1[] = {
    "(1x,2i5,3f10.2)"
};
    WRITE(OUTPUT,FMT,F1,1,INT4,i,INT4,j,REAL4,a,REAL4,b,REAL4,c,0);
    WRITE(OUTPUT,LISTIO,INT4,i,INT4,j,REAL4,a,REAL4,b,REAL4,c,0);
}
```

In this translation the PROMULA.FORTRAN runtime library is used to perform the writes. The precise layout of the records produced conform exactly to the FORTRAN standard specification. The C output using the Yf flag looks as follows.

```
void demo()
{
static int i,j;
static float a,b,c;
    printf(" %5ld%5ld%10.2f%10.2f%10.2f\n",i,j,a,b,c);
    printf("%12ld%12ld%16.6E%16.6E%16.6E\n",i,j,a,b,c);
}
```

In this translation the FORMAT string has been converted into an equivalent `printf` style string. In the list-directed form `printf` elements have been selected to correspond to each element in the list. Note that these elements are in the FORTRAN dialect definition file and may be changed by you.

The C `printf` conventions are reasonable for simple formatting operations; however, they lack the power of the FORTRAN FORMAT string. The output produced by the Yf flag is an approximation only of that produced by the runtime library. In particular, it does not conform to the FORTRAN standard specification. Use the Yf flag only if you are moving away from FORTRAN completely and if your application does not require particular formatting conventions.

2.43 Initialization Check for Auto Variables — Xi, Yi

Traditional FORTRAN programs assume that local variables maintain their values through calls to a given subprogram. As a result of this fact the default treatment of all variables by PROMULA is to make them static. See the section on the storage allocation switches for more information on variable allocation. For users who wish to declare variables auto, the Yi flag can be used to check for variables that are used before they are initialized. It is this class of variables that must be static.

In essence, the Yi flag tells PROMULA to override an auto storage allocation for a variable whose value appears to be used before it is changed. The actual algorithm used is simplistic and does not pretend to catch all problem variables.

As a simple example of the use of this variable, consider the following code in which the variable `I` is being used without being explicitly set.

```
      SUBROUTINE DEMO
      J = 5
      K = I + J
      WRITE(*,*) I,J,K
      RETURN
      END
```

The translation of this code using SS and SA0 is as follows.

```
void demo()
{
auto long j,k,i;
    j = 5;
    k = i+j;
    WRITE(OUTPUT,LISTIO,INT4,i,INT4,j,INT4,k,0);
    return;
}
```

The variable `i` is auto. In all likelihood codes having structures such as this would not produce the correct result with the above translation. Using the Yi flag produces the following output, in which `i` has been forced to static.

```
void demo()
{
auto long j,k;
static long i;
    j = 5;
    k = i+j;
    WRITE(OUTPUT,LISTIO,INT4,i,INT4,j,INT4,k,0);
    return;
}
```

The Yi flag is not a complete solution to the problem of uninitialized variables and variables that are to retain their values; however, it can help to avoid problems.

2.44 DO Loop Counter Reduction Optimization — Xl, Yl

As is also discussed in the section on array subscript expressions and in the section on the DO statement, when the only purpose of a DO loop counter is to subscript arrays within the loop, that counter can be reduced to simplify the array subscript expressions. The Yl switch turns this optimization on and is the default for all biases. The Xl switch turns this optimization off.

As an example of this optimization, consider the following simple FORTRAN code which performs a matrix multiplication.

```
      SUBROUTINE MATMUL(A,B,C)
      DIMENSION A(10,15),B(15,20),C(10,20)
      DO 15 I = 1,10
      DO 15 J = 1,20
      C(I,J) = 0.0
      DO 10 K = 1,15
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
   10 CONTINUE
   15 CONTINUE
      RETURN
      END
```

Notice that in FORTRAN subscripts normally start at 1; therefore, the loop variables in the above example also all start at 1. Now the default C output for this example, with Yl active, is shown below.

```
void matmul(a,b,c)
float a[15][10],b[20][15],c[20][10];
{
static int i,j,k;
    for(i=0; i<10; i++) {
        for(j=0; j<20; j++) {
            c[j][i] = 0.0;
            for(k=0; k<15; k++) {
                c[j][i] = c[j][i]+a[k][i]*b[j][k];
            }
        }
    }
    return;
}
```

In C subscripts always start at 0; therefore, the DO loop counters themselves have been changed to start at zero. The listing below shows the C output with Xl active.

```
void matmul(a,b,c)
float a[15][10],b[20][15],c[20][10];
{
static int i,j,k;
    for(i=1; i<=10; i++) {
        for(j=1; j<=20; j++) {
            c[j-1][i-1] = 0.0;
            for(k=1; k<=15; k++) {
                c[j-1][i-1] = c[j-1][i-1]+a[k-1][i-1]*b[j-1][k-1];
            }
        }
    }
    return;
}
```

In this form, the loop variables have their original ranges; however, each reference to those variables must be decremented by one to compensate for the difference in C and FORTRAN subscripting conventions.

2.45 Subprogram Argument Type Checking — Xp, Yp

Though the practice is never desirable, most FORTRAN compilers allow the user to pass variables of different types to the same subprogram parameter. PROMULA follows the normal practice and does not perform checks to make certain that all parameters are consistent. The YP switch turns this checking on. If you have a program that is carefully written, or if you want to check that all parameters are passed consistently, then the Yp flag performs this service.

Consider the following simple subprogram, in which the third parameter to function `demo` is REAL*4 in one instance and INTEGER*4 in another.

```
      SUBROUTINE TEST
      CALL DEMO(I,J,A)
      CALL DEMO(I,J,K)
      END
```

The default C output for this subprogram, with Xp active, is simply as follows.

```
void test()
{
extern void demo();
static long i,j,k;
static float a;
    demo(&i,&j,&a);
    demo(&i,&j,&k);
}
```

In one place `demo` is called with a pointer to a long and in the other with a pointer to a float. Alternatively, with Yp active, the following is produced.

```
3: test.for: E247: The argument K of type integer*4 has been entered where a call-by-
                   reference argument of type real*4 is required.
```

Alternatively, if the error checking level is set to 1 or above with the EL switch, the following warning is produced without use of the Yp switch.

```
3: test.for: W814: The argument K of type integer*4 has been entered where an
                   argument of type real*4 has been used.
```

2.46 Single Precision Real Arithmetic — Xr, Yr

Though the designers of C made a few strange decisions, their only real mistake was to omit single precision real arithmetic. Even now the major barrier to the acceptance of PROMULA as a "pure" compiler is that its benchmarks on single precision arithmetic programs are often slow. Fortunately, the newer ANSI C compilers do support single precision real arithmetic. Unfortunately, they require that single precision real constants have an "F" appended to distinguish them from double precision constants. Most older C compilers consider this "F" to be a syntax error; therefore, you must tell PROMULA that your C compiler is capable of performing single precision real arithmetic. The default Xr switch assumes that all single precision real arithmetic will be performed as double precision, and that all single precision reals will be promoted to double when passed by value.

The Yr flag, on the other hand, treats single precision real arithmetic completely separately from double. No automatic promotion is assumed. In addition, single precision real constants have an "F" appended.

2.47 Subscript Pointer Notation — Xs, Ys, Ysv, Ysf

There is a difficult translation issue associated with array subscript expressions. In FORTRAN an array is a set of elements identified by a single name. The dimension of the array merely specifies how many elements there are and how they are organized. The elements in an array are always stored contiguously in memory in row major order. That is, the leftmost subscript varies the fastest as elements are accessed in storage order. When a FORTRAN array is defined, its minimum and

maximum subscript bounds may be any value, positive or negative, just so long as the minimum bound is less than the maximum bound. If no minimum bound is specified, a minimum of one is assumed.

In C there is a strong relationship between pointers and arrays. Though the array notation and pointer notation may be mixed fairly freely, the pointer version will, in general, be more efficient. Multidimensional arrays are stored contiguously in memory in column major order. That is, the rightmost subscript varies the fastest as elements are accessed in storage order. When defining an array, the user may only specify its size, the minimum bound is always assumed to be zero.

As can be seen from the above, there are considerable differences in the manner in which arrays are treated in the two languages. For the C and optimized biases, because C programmers are familiar with pointer notation and because many optimizations can be achieved when using pointer notation, all FORTRAN arrays are defined in C as simple one dimensional vectors and all array references are done using the pointer notation. Alternatively, for the FORTRAN bias C brackets notation is used whenever possible; however, the order of the subscripts must be reversed in the C output. In addition the "s" — for subscript — optimization switch is provided. Using this switch, the user may specify his own preference on top of his overall bias specification.

The following shows a sample set of fixed array definitions along with some basic subscript expressions as translated via the Xs and Ys switches.

```
      SUBROUTINE DEMO
      DIMENSION A(10),B(6,7),C(3,4,5)
     +  ,D(2,3,4,5)
      A(3) = 0
      B(2,4) = 1
      C(3,4,5) = 2
      D(1,1,1,1) = 3
      A(I) = 4
      B(2,J) = 5
      C(I,2,K) = 6
      D(I,2,K,L) = 7
      END
```

Intermediate C output using the Ys switch is as follows.

```
void demo()
{
static int i,j,k,l;
static float a[10],b[42],c[60],d[120];
    *(a+2) = 0;
    *(b+19) = 1;
    *(c+59) = 2;
    *d = 3;
    *(a+i-1) = 4;
    *(b+1+(j-1)*6) = 5;
    *(c+i-1+(1+(k-1)*4)*3) = 6;
    *(d+i-1+(1+(k-1+(l-1)*4)*3)*2) = 7;
}
```

Intermediate C output using the default Xs switch is as follows.

```
void demo()
{
static float a[10],b[7][6],c[5][4][3],d[5][4][3][2];
static long i,j,k,l;
    a[2] = 0;
    b[3][1] = 1;
    c[4][3][2] = 2;
    d[0][0][0][0] = 3;
    a[i-1] = 4;
    b[j-1][1] = 5;
```

```
    c[k-1][1][i-1] = 6;
    d[l-1][k-1][1][i-1] = 7;
}
```

Notice first in this example that, though both the FORTRAN source and the Xs switch translation show multidimensional arrays, the order of the array sizes is reversed in the translation. This is necessary to maintain the same relative positioning of the elements within the arrays. In the Ys translation, arrays are all shown with a single dimension equal to the total size of each array. Thus, D shows 120 values which is

```
2 * 3 * 4 * 5.
```

In both the translations the actual subscript values are reduced by one to reflect the fact that C subscripts begin at zero, while FORTRAN begins at one. In the Ys case, the subscript array references are translated using pointer notation, with the calculation reduced to its simplest form. Thus, `C(3,4,5)` becomes `*(c+59).` Also, subscript calculations are performed using the FORTRAN convention. This ensures that any games played in the programs with equivalencing or with varying COMMON layouts will work exactly as they worked in FORTRAN. Using a C compiler which does constant reduction one would expect to have the same code produced by both of the above translations.

To demonstrate the subscript simplification process, the listing below shows the same Ys bias translation, but with the "constant reductions optimizations" turned off.

```
void demo()
{
static int i,j,k,l;
static float a[10],b[42],c[60],d[120];
    *(a+3-1) = 0;
    *(b+2-1+(4-1)*6) = 1;
    *(c+3-1+(4-1+(5-1)*4)*3) = 2;
    *(d+1-1+(1-1+(1-1+(1-1)*4)*3)*2) = 3;
    *(a+i-1) = 4;
    *(b+2-1+(j-1)*6) = 5;
    *(c+i-1+(2-1+(k-1)*4)*3) = 6;
    *(d+i-1+(2-1+(k-1+(l-1)*4)*3)*2) = 7;
}
```

Two additional subscripting options are provided for those who want to write their own approach to subscripting — the Ysv and Ysf flags. The listing below shows the C intermediate output with the Ysv switch active.

```
void demo()
{
static int i,j,k,l;
static float a[10],b[42],c[60],d[120];
    *_aref(a,2) = 0;
    *_aref(b,1,6,3) = 1;
    *_aref(c,2,3,3,4,4) = 2;
    *_aref(d,0,2,0,3,0,4,0) = 3;
    *_aref(a,i-1) = 4;
    *_aref(b,1,6,j-1) = 5;
    *_aref(c,i-1,3,1,4,k-1) = 6;
    *_aref(d,i-1,2,1,3,k-1,4,l-1) = 7;
}
```

In this output a generalized symbol `_aref` is provided followed by a pointer to the start of the array followed by a sequence of ordered pairs containing the subscript value and the size of the dimension associated with that value. The user can then either write a macro in fortran.h or a function to perform the actual subscript calculations.

The Ysf flag uses the same approach as Ysf except that the subscript values are not offset to make them conform to the C convention. The following shows the output for Ysf.

```
void demo()
{
static int i,j,k,l;
static float a[10],b[42],c[60],d[120];
    *_aref(a,3) = 0;
    *_aref(b,2,6,4) = 1;
    *_aref(c,3,3,4,4,5) = 2;
    *_aref(d,1,2,1,3,1,4,1) = 3;
    *_aref(a,i) = 4;
    *_aref(b,2,6,j) = 5;
    *_aref(c,i,3,2,4,k) = 6;
    *_aref(d,i,2,2,3,k,4,l) = 7;
}
```

Notice that the subscript values are now exactly as entered in the original FORTRAN code. Again as with Ysv it is up to the user to provide an appropriate implementation of the `_aref` function or macro.

2.48 Unformatted Write Optimization — Xu, Yu

It is a rather strange fact about FORTRAN that the only file type that requires special internal formatting, above that used by the normal I/O system, is the unformatted file. The problem is that unformatted files receive variable length records. It is up to the runtime library to ensure that records are not overflowed and to ensure that each new read begins at the start of a record. To make this possible the runtime library, when it writes unformatted records, must put a length value at the front and back of each record. This convention creates a file structure which is efficient to read, backspace, and in general use. To write the file, however, requires repositioning the file twice for each record written. To avoid this repositioning overhead, a flag is available that tells PROMULA to compute the lengths of records before they are written.

The following is a simple FORTRAN code which performs a series of unformatted writes to unit 1.

```
      SUBROUTINE DEMO(A,B,C,NR,NC)
      DIMENSION A(NR),B(NC),C(NR,NC)
      WRITE(1) A
      WRITE(1) (A(I),I=1,6)
      WRITE(1) B,C
      RETURN
      END
```

The default translation, under Xu, is as follows.

```
void demo(a,b,c,nr,nc)
int *nr,*nc;
float a[],b[],*c;
{
static int i;
    WRITE(1,REAL4,a,*nr,0);
    WRITE(1,MORE);
    for(i=0; i<6; i++) {
        WRITE(REAL4,&a[i],1,MORE);
    }
    WRITE(0);
    WRITE(1,REAL4,b,*nc,REAL4,c,*nr**nc,0);
    return;
}
```

The listing is easy to read and corresponds to the original FORTRAN quite closely. When executed, it will be reasonably efficient, but each write will require that the file be repositioned twice.
Using the Yu flag the following C output is obtained.

```
void demo(a,b,c,nr,nc)
int *nr,*nc;
```

```
float a[],b[],*c;
{
static int T1,i,T2,T3;
    T1 = *nr*4;
    fiouwl(&T1);
    WRITE(1,REAL4,a,*nr,0);
    T2 = 24;
    fiouwl(&T2);
    WRITE(1,MORE);
    for(i=0; i<6; i++) {
        WRITE(REAL4,&a[i],1,MORE);
    }
    WRITE(0);
    T3 = *nc*4+*nr**nc*4;
    fiouwl(&T3);
    WRITE(1,REAL4,b,*nc,REAL4,c,*nr**nc,0);
    return;
}
```

In this output, temporary variables are computed which contain the lengths of the records to be written. These values are then passed to a runtime function `fiouwl` which sets up the length value at the front of the record before it is written. As a result, repositioning of the file is not required. The actual C output is, of course, more difficult to read; therefore, it is suggested that this flag only be used if unformatted output speed is critical.

2.49 Subprogram Call-by-Value Arguments — Xv, Yv

In FORTRAN all parameters are passed by address. This convention generates inefficient code for simple values and makes passing constants and expressions messy, insofar as the translation is concerned. Nevertheless, to ensure that PROMULA.FORTRAN always produces correct results, the default convention used is to pass all arguments by address. The Yv switch can be used to allow for call-by-value.

Note that explicit function prototypes can always be provided via the configuration file. These prototypes may specify call-by-value parameters on an individual function level which override the default setting of this switch.

2.50 Dollar Signs as Initial Symbols in Identifiers — X$, Y$

By default PROMULA accepts $ (dollar sign) as a valid character within identifiers. They may occur anywhere that an alphabetic character may appear. Unfortunately, some dialects of FORTRAN use the $ to mark multiple return statement labels. For these dialects the $ may not be allowed to occur in the initial position. The X$ switch disallows $ in the initial position of identifiers. The default Y$ switch allows them to occur anywhere.

2.51 Location of FORTRAN Files to be Included — Zname

It is often desirable to have include files that are referenced from within FORTRAN programs stored in some other directories. The Zname flag can be used to specify directories to be searched for include files. The syntax of name varies from operating system to operating system, but should be the same as the syntax used to construct the PATH variable used to locate the executable for PROMULA.FORTRAN itself.

If the first character of `name` is a `#`, then the remainder of `name` is assumed to be the name of an environment variable which contains the include file search paths.

2.52 Project Processing — #project

In addition to processing independent single source files, it is often desirable to process groups of related multiple files as a single project. This is done via a project file which has the following syntax:

```
PROJECT:   project_name
SOURCE:    file1.for,
```

```
                file2.for,
                .
                .
                .
                fileN.for
```

where:

`project_name` is the name of the project
`fileI.for` are the names of the files that make up the project

To process a project, the following command line is used:

```
    pfc #project  options
```

where:

`project` is the name of the project file. If no extension is provided `.prj` is assumed.

`options` contains any other command line switches to be used.

The special character # differentiates a project file from a single source file. To avoid conflicts with other uses of this character in some operating environments (e.g., the Korn shell in UNIX), the alternative character `@` may be used to mark the project file name.

The processing of the project file will produce the following three kinds of files:

1. The translated FORTRAN files named `fileI.c`
2. A global prototypes file, `project_name.h`
3. A global variables file, `project_name.c`

To avoid conflicts, make sure that the name of the project is different than the names of all the project member files.

3. CONFIGURATION FILE

PROMULA translates various dialects of FORTRAN source code to a variety of possible C source code outputs. It has a powerful dialect processing component for controlling both the syntax of the FORTRAN dialect it accepts as input and the form of the C source code it produces as output. This is accomplished via an external configuration file which contains detailed statements for customizing the translation process. These statements include the following:

SWITCHES — specifies command line switches
COMMENTS — controls format of comments
PATHNAMES — controls location and naming of include filenames
RESTRUCTURE — restructures variables in the source code
KEYWORDS — changes output keywords and pattern strings
PRAGMA — configures special comments into C pragmas
$ — controls treatment of the dollar sign
Function prototypes — control the translation of functions and their arguments

The configuration file is used with the R option on the pfc command line, as follows:

```
pfc file_name[.for] Rconf[.cnf]
```

The rules for writing directives in configuration files are described in the context of the following sections.

Suffice to say here that the Rname switch will read configuration information from the files named. The Rname switch may occur multiple times to allow the reading of multiple files. If there is no extension supplied with the first file, an extension of `.cnf` will be assumed. If there is no extension supplied for additional files, an extension of `.pro` is assumed. The reason for this is that typically, when multiple files are read, the first contains the configuration information; while the additional files contain function prototype information.

It should be noted that configuration files may contain command line switch information; thus, a common use of a configuration file is to supply commonly used switches independently of the command line.

3.1 The Configuration SWITCHES Statement

The PROMULA translation process is controlled via many command line switches. These switches give the user easy access to the translation process; however, there are often many needed. To simplify the organization of complex groups of switches, the configuration file contains a SWITCHES statement.

Syntax:

```
SWITCHES   s1 ... sn
```

Where:

`s1 ... sn` are any valid sequence of command line switches entered exactly as they would be entered on the command line

There may be as many SWITCHES statements as desired in a given configuration file; however, the set of SWITCHES statements must always be the first set of statements in the configuration file. If multiple configuration files are being processed, a SWITCHES statement may only appear in the first such file. Note that the Rname switch may appear within a SWITCHES statement.

The following is an example of a SWITCHES statement.

```
SWITCHES Y3 P+4096 P+8192 CF11 Il Mvax CHv Ga Yd*? QH1000
```

Note that the SWITCHES statement often interacts with other statements following it in the configuration file. Many examples of this interaction appear in the following discussion of the other statements in the configuration file.

3.2 The Configuration COMMENTS Statement

Many FORTRAN programs have relatively complicated conventions for marking comments of various types. These conventions often do not work well when transformed directly to C. As has been mentioned elsewhere in this manual, the hardest single aspect of FORTRAN codes to deal with in translation is the commenting conventions.

In general, the statements of a FORTRAN program can be divided into two groups:

(1) those that define the symbols to be used — referred to as data definition language or DDL;

(2) those that specify the actual operations to be performed — referred to as data management language or DML.

Insofar as comments are concerned, the translation of the DML is straightforward since the ordering of the DML in the target language is typically the same as that in the source language. For the DDL, however, the ordering of symbol definitions is often quite different in the target language; therefore, the comments must also often be rearranged. The algorithms for associating comments with symbols are complex and by no means perfect.

The COMMENTS statement itself does not deal with the ordering of comments but rather deals with the source and target form of the individual comment lines themselves.

Syntax:

```
COMMENTS
DDL-specification
DML-specification
END
```

Where:

`DDL-specification` says how comment lines within the DDL are to be interpreted and translated
`DML-specification` says how comment lines within the DML are to be interpreted and translated

Each specification consists of three sections. Within each section, the fields are as follows:

Section	Content	Description
1	0 1 "str"	Do not precede comment blocks with a standard header Precede comment blocks with the indicated header
2	"s1" "s2"...	A series of individual comment introducers and their replacements in C.
3	0 1 "str"	Do not follow comment blocks with a standard trailer Follow comment blocks with the indicated trailer.

As an example, consider the following FORTRAN code.

```
        FUNCTION MODI (ARG_SWAB_FLAG, ARG_XLIT_FLAG)
!!
!!      PURPOSE:        General purpose initialization for MODI_* routines.
!!
!!      PARAMETERS:     LOGICAL ARG_SWAB_FLAG (R)
```

```
!!                      LOGICAL ARG_XLIT_FLAG (R)
!!
!!      RETURN VALUE:   None.
!!
!!      SIDE EFFECTS:   None.
!!
!!
! Incorporates features to support portable-LIO file handling (8-Apr-1986).

        IMPLICIT INTEGER (A-Z)

!=======================================================================
!                         Initialization entry

! Initialization-- calculate some useful things.

        NONINPBASE = HD_QPIS + HD_QPSUPIS
        FIRSTNONINP = NONINPBASE + 1  ! First non-input node

! Determine practical maximum for
! no. of strings we can hash (using "PUTSTR" entry)--
! (Knuth's vol.3 says that the avg. no. collisions =
! 1/2 * (1 + 1/R), where "R" is HASH table density);
! it might be nice to stop at 98% full (R = 0.98),
! since this translates to an average of 25.5 collisions
! per string entered.

        HASHMAX = HD_HASHVAL - (2 * HD_HASHVAL) / 100

! Save local values of SWAB and XLIT flags, and convert blank-string to
! match the language (EBCDC or ASCII) of the model-file-CREATOR.

        SWAB_FLAG = ARG_SWAB_FLAG
        XLIT_FLAG = ARG_XLIT_FLAG
        IF (XLIT_FLAG .AND. .NOT. BLANKS_XLIT_YET) THEN
             CALL TRLITO (BLANK_BYTES, 4)
             BLANKS_XLIT_YET = .TRUE.
        ENDIF
        RETURN
        END
```

Within this code the `!` is being used to mark the comments and multiple occurrences of this symbol are used for emphasis. The default translation looks as follows.

```
long modi(arg_swab_flag,arg_xlit_flag)
/*
!
!      PURPOSE:        General purpose initialization for MODI_* routines.
!
!      PARAMETERS:     LOGICAL ARG_SWAB_FLAG (R)
!                      LOGICAL ARG_XLIT_FLAG (R)
!
!      RETURN VALUE:   None.
!
!      SIDE EFFECTS:   None.
!
!
 Incorporates features to support portable-LIO file handling (8-Apr-1986).
*/
long *arg_swab_flag,*arg_xlit_flag;
{
extern void trlito();
```

```
static long K1 = 4;
static long modi,noninpbase,hd_qpis,hd_qpsupis,firstnoninp,hashmax,hd_hashval,
    swab_flag,xlit_flag,blanks_xlit_yet,blank_bytes;
/*
=============================================================================
                          Initialization entry
 Initialization-- calculate some useful things.
*/
    noninpbase = hd_qpis+hd_qpsupis;
    firstnoninp = noninpbase+1L;   /* First non-input node*/
/*
 Determine practical maximum for
 no. of strings we can hash (using "PUTSTR" entry)--
 (Knuth's vol.3 says that the avg. no. collisions =
 1/2 * (1 + 1/R), where "R" is HASH table density);
 it might be nice to stop at 98% full (R = 0.98),
 since this translates to an average of 25.5 collisions
 per string entered.
*/
    hashmax = hd_hashval-2L*hd_hashval/100L;
/*
 Save local values of SWAB and XLIT flags, and convert blank-string to
 match the language (EBCDC or ASCII) of the model-file-CREATOR.
*/
    swab_flag = *arg_swab_flag;
    xlit_flag = *arg_xlit_flag;
    if((xlit_flag & fifi4log(!blanks_xlit_yet))) {
        trlito(&blank_bytes,&K1);
        blanks_xlit_yet = fifi4log((long)TRUE);
    }
    return modi;
}
```

Under this default the actual comment symbol is removed and the remainder of the comment is left alone. Each block of comments is preceded by the C comment delimiter `/*` and is followed by the delimiter `*/`. The resulting appearance of the comments is very different from that of the source.

The COMMENTS statement shown below

```
COMMENTS
0  "!!" " **" "!" " *"  0
0  "!!" " **" "!" " *"  0
END
```

tells PROMULA to treat DML and DDL comments in the same manner. The first section of each specification says not to have a separate header for the comment blocks. The second section establishes the following correspondence between source and target comment introducers.

Source	Target
!!	**
!	*

The third section says not to have a separate trailer for each comment block.

Using this COMMENTS statement in a configuration file produces the following result.

```
long modi(arg_swab_flag,arg_xlit_flag)
/**
 **      PURPOSE:        General purpose initialization for MODI_* routines.
 **
```

```
 **      PARAMETERS:     LOGICAL ARG_SWAB_FLAG (R)
 **                      LOGICAL ARG_XLIT_FLAG (R)
 **
 **      RETURN VALUE:   None.
 **
 **      SIDE EFFECTS:   None.
 **
 **
 * Incorporates features to support portable-LIO file handling (8-Apr-1986).*/
long *arg_swab_flag,*arg_xlit_flag;
{
extern void trlito();
static long K1 = 4;
static long modi,noninpbase,hd_qpis,hd_qpsupis,firstnoninp,hashmax,hd_hashval,
    swab_flag,xlit_flag,blanks_xlit_yet,blank_bytes;
/*=========================================================================
 *                          Initialization entry
 * Initialization-- calculate some useful things.*/
    noninpbase = hd_qpis+hd_qpsupis;
    firstnoninp = noninpbase+1L;   /* First non-input node*/
/* Determine practical maximum for
 * no. of strings we can hash (using "PUTSTR" entry)--
 * (Knuth's vol.3 says that the avg. no. collisions =
 * 1/2 * (1 + 1/R), where "R" is HASH table density);
 * it might be nice to stop at 98% full (R = 0.98),
 * since this translates to an average of 25.5 collisions
 * per string entered.*/
    hashmax = hd_hashval-2L*hd_hashval/100L;
/* Save local values of SWAB and XLIT flags, and convert blank-string to
 * match the language (EBCDC or ASCII) of the model-file-CREATOR.*/
    swab_flag = *arg_swab_flag;
    xlit_flag = *arg_xlit_flag;
    if((xlit_flag & fifi4log(!blanks_xlit_yet))) {
        trlito(&blank_bytes,&K1);
        blanks_xlit_yet = fifi4log((long)TRUE);
    }
    return modi;
}
```

This form has the same effect as the FORTRAN original; however, for both the DDL and DML comments, the lack of a trailer makes the code hard to read, and for the DML comments a header seems appropriate as well.

```
COMMENTS
0          "!!" " **" "!" " *"   1 " */"
1 "/*"     "!!" " **" "!" " *"   1 " */"
END
```

Using this specification produces a much nicer looking translation which still seems to capture the flavor of the original.

```
long modi(arg_swab_flag,arg_xlit_flag)
/**
 **      PURPOSE:        General purpose initialization for MODI_* routines.
 **
 **      PARAMETERS:     LOGICAL ARG_SWAB_FLAG (R)
 **                      LOGICAL ARG_XLIT_FLAG (R)
 **
 **      RETURN VALUE:   None.
 **
 **      SIDE EFFECTS:   None.
 **
 **
 * Incorporates features to support portable-LIO file handling (8-Apr-1986).
```

```
 */
long *arg_swab_flag,*arg_xlit_flag;
{
extern void trlito();
static long K1 = 4;
static long modi,noninpbase,hd_qpis,hd_qpsupis,firstnoninp,hashmax,hd_hashval,
    swab_flag,xlit_flag,blanks_xlit_yet,blank_bytes;
/*
 *=========================================================================
 *                          Initialization entry
 * Initialization-- calculate some useful things.
 */
    noninpbase = hd_qpis+hd_qpsupis;
    firstnoninp = noninpbase+1L;   /* First non-input node*/
/*
 * Determine practical maximum for
 * no. of strings we can hash (using "PUTSTR" entry)--
 * (Knuth's vol.3 says that the avg. no. collisions =
 * 1/2 * (1 + 1/R), where "R" is HASH table density);
 * it might be nice to stop at 98% full (R = 0.98),
 * since this translates to an average of 25.5 collisions
 * per string entered.
 */
    hashmax = hd_hashval-2L*hd_hashval/100L;
/*
 * Save local values of SWAB and XLIT flags, and convert blank-string to
 * match the language (EBCDC or ASCII) of the model-file-CREATOR.
 */
    swab_flag = *arg_swab_flag;
    xlit_flag = *arg_xlit_flag;
    if((xlit_flag & fifi4log(!blanks_xlit_yet))) {
        trlito(&blank_bytes,&K1);
        blanks_xlit_yet = fifi4log((long)TRUE);
    }
    return modi;
}
```

This sample is an actual fragment. We did not create it to make this feature look good. If the source has a consistent commenting style, then an equivalent one for the target language can be created using this statement. It is, of course, the user's responsibility to ensure that proper comments are being written.

3.3 The Configuration PATHNAMES Statement

A major problem faced in the migration of source codes from one environment to a new environment has to do with the accessing of include files. There are three problem areas:

(1) Source pathnames often include dollar signs and possibly other characters such as slashes which cannot be easily used under target systems.

(2) The directory separation characters are not the same on different systems.

(3) The organization and search rules used on the source system might not be the ones needed for the new environment.

These problems are dealt with explicitly in PROMULA via the PATHNAMES statement in the configuration file. Though it may not be feasible for all users to use the same PATHNAMES specifications, no user need physically make any source changes to the INCLUDE or INSERT statements in his source codes. This section describes how a particular pathname translation scheme can be established for a given set of conventions.

The PATHNAME statement is entered into the configuration file. It describes the path and filename conventions to be used on the target platform and how these conventions are to be obtained from the source specifications. The approach taken is to describe how source pathnames are to be "translated" into target pathnames.

Syntax:

```
PATHNAMES dirchar [REPLACE "s1t1s2t2..."]
                  [LOWER | UPPER]
                  [PREFIX "tname"]
                  [EXCLUDE]
                  [TERMINATION tc]

sname(1) tname(1)
.
.
.
sname(n) tname(n)
END
```

Where:

`dirchar` is the directory separation character in the source pathname

`s1t1s2t2...` are a sequence of character pairs

`tname` is a target language pathname or pathname prefix

`sname` is a source language pathname or pathname prefix

`tc` is a pathname termination character used in the source pathname

The required `dirchar` specification specifies the character used to separate the pathname components in the original source codes. This character is replaced by the equivalent character in the target pathnames. For example moving from PRIME FORTRAN to UNIX, a `<` character would be replaced by a `/` character. Moving from MS-DOS to UNIX would replace a `\` character with a `/` character.

The optional REPLACE parameter specifies additional characters to be replaced in the source names. As many pairs of characters as are needed may be included. The standard PRIME language description, for example, contains the following specification for this option:

```
REPLACE "$_"
```

This causes all dollar signs in the source pathnames to be replaced by underscores.

The TERMINATION character is provided for source systems such as VMS which allow disposition information to be appended to the filename preceded by a slash. All of this information must be ignored in most target systems.

The mutually exclusive and optional UPPER, LOWER parameters specify that all alphabetic characters in pathnames should be converted to upper- or lower-case respectively. Since some systems' pathnames are case sensitive, while others are not, it is important to specify one of these options. For example, though most PRIME pathnames are shown in uppercase, most transfer programs create lowercase names when transferring files to UNIX; therefore, the standard PRIME language description contains a specification of LOWER for this option.

For initial testing and use of PROMULA for particular small projects, the simplest approach is simply to move all source files — including the INCLUDE files — into the user's local directory. To do this PROMULA must be told to ignore all directory information in the source pathnames. Under this alternative all characters up to and including the last occurrence of the directory component separations character are stripped from the source pathname. This is achieved via the EXCLUDE option.

A possible alternative structure for the INCLUDE files for a UNIX implementation might be to copy all of these files into some subdirectory where they would retain the same relative structure as they had on the source system. The PREFIX `pname` option allows a directory specification to be added to the front of all source pathnames.

Another alternative might be to copy all include files into a single subdirectory with no additional structure. This effect can be achieved by using PREFIX in conjunction with the EXCLUDE option. All source structure would be excluded and then would be replaced by the desired target subdirectory name.

In some cases, no generic translation scheme will work. Certain names might have to be changed on an individual basis. The final list of `sname`, `tname` pairs achieves this end. Each pathname is first translated using the generic specifications on the PATHNAME statement itself. The resultant pathnames are compared with the `snames` in the list. If the first `n` characters of a pathname match the `n` characters of an `sname`, then those `n` characters are stripped and the associated `tname` is added to the front of the name.

As can be seen from the above, it will be necessary to organize the INCLUDE files in the new environment. Once that organization has been completed, the PATHNAMES component of the configuration file can be used to describe that structure. No changes need be made in the FORTRAN source code INCLUDE and INSERT statements.

3.4 The Configuration RESTRUCTURE Statement

The RESTRUCTURE statement in the configuration file is used in contexts where the symbol table must be changed in order to make a program work. Typically, it is used when a program is being moved between platforms with different word sizes.

Syntax:

```
RESTRUCTURE
ident   type
.
.
END
```

Where:

`ident`	is the identifier of a variable in the program being processed.
`type`	is the FORTRAN type to be assigned to that variable.

To understand the use of the RESTRUCTURE statement, consider the following simple FORTRAN program written for a platform with 36 bit words and 6 bit characters.

```
      PROGRAM TEST3
      DIMENSION IALPHA(2)
      DATA IALPHA/6HHello ,6HWorld./
    1 FORMAT(1X,2A6)
      PRINT 1,IALPHA
      STOP
      END
```

In this program the INTEGER array `IALPHA` is being used to contain character data. This is a standard convention in many FORTRAN dialects still in use today. The default translation for this code is a mess.

```
void main(argc,argv)
int argc;
char* argv[];
{
static long ialpha[2];
static int ftnsiz[] = {1,1,2};
```

```
static namelist DATAVAR[] = {
"ialpha",ialpha,5,ftnsiz
};
static char *DATAVAL[] = {
"$DATAVAR",
"ialpha='Hell','o',",
"$END"
};
static char* F1[] = {
    "(1x,2a6)"
};
    ftnini(argc,argv,NULL);
    fiointu((char*)DATAVAL,0,2);
    fiornl(DATAVAR,1,NULL);
    WRITE(OUTPUT,FMT,F1,1,DO,2,INT4,ialpha,0);
    STOP(NULL);
}
```

Since the variable is declared as an INTEGER, runtime initialization is needed. In addition, since PROMULA assumes no more that 4 characters per word the phrase `World` is thrown away. Using the QW3606 switch improves the situation slightly; however, the result will not run on a 32 bit platform

```
void main(argc,argv)
int argc;
char* argv[];
{
static long ialpha[2];
static int ftnsiz[] = {1,1,2};
static namelist DATAVAR[] = {
"ialpha",ialpha,5,ftnsiz
};
static char *DATAVAL[] = {
"$DATAVAR",
"ialpha='Hello','World.',",
"$END"
};
static char* F1[] = {
    "(1x,2a6)"
};
    ftnini(argc,argv,NULL);
    fiointu((char*)DATAVAL,0,2);
    fiornl(DATAVAR,1,NULL);
    WRITE(OUTPUT,FMT,F1,1,DO,2,INT4,ialpha,0);
    STOP(NULL);
}
```

because the long variable `ialpha` is now expected to contain 6 characters. We need to tell the translator that this variable is really a CHARACTER*6. The following configuration file does this.

```
SWITCHES QW3606
RESTRUCTURE
IALPHA  CHARACTER*6
END
```

Using this configuration file, the following "correct" translation is produced.

```
void main(argc,argv)
int argc;
char* argv[];
{
static char ialpha[12] = {
```

```
    'H','e','l','l','o',' ','W','o','r','l','d','.'
};
static char* F1[] = {
    "(1x,2a6)"
};
    ftnini(argc,argv,NULL);
    WRITE(OUTPUT,FMT,F1,1,DO,2,STRG,ialpha,6,0);
    STOP(NULL);
}
```

3.5 The Configuration KEYWORDS Statement

Most of the actual text output of PROMULA is formed by combining sequences of keywords or keyword strings. These strings can be changed by the user, although if they are changed the user must make certain that the result is still compilable. Most KEYWORD changes require an equivalent change in the fortran.h header file. Each KEYWORD can be referenced either by its current form or by its identification number.

The keywords themselves are of two types:

(1) simple strings
(2) pattern strings

The table below shows the simple strings within the keyword table. These appear in the output of PROMULA exactly as shown in the appropriate places.

Idn	Keyword	Description of use in C
1	void	equivalent of SUBROUTINE
2	short	equivalent of INTEGER*2
3	double	equivalent of REAL*8
4	unsigned short	equivalent of LOGICAL*2
5	char	equivalent of INTEGER*1
6	long	equivalent of INTEGER*4
7	float	equivalent of REAL*4
8	unsigned long	equivalent of LOGICAL*4
9	unsigned char	equivalent of LOGICAL*1
10	double	equivalent of REAL*16
11	dcomplex	equivalent of DOUBLE COMPLEX
12	complex	equivalent of COMPLEX
13	char	equivalent of CHARACTER
14	static	static keyword
15	auto	auto keyword
16	extern	extern keyword
17	typedef	typedef keyword
18	struct	struct keyword
19	ftnadr	array that contains assigned FORMAT pointers
20	ftnalloc	allocates memory for dynamic variables
21	ftndim	array for dimension values for namelist
22	ftnlen	number of lines in assigned FORMATs
23	ftnfree	frees memory for dynamic variables
24	ftnsiz	array for dimension values for runtime data
25	namelist	structure name for namelist
26		declaration which precedes includes
27	ftnini(argc,argv,NULL);	runtime initialization function call
28	vmsopn(argc,argv);	runtime initialization call for virtual memory
29	void main(argc,argv)	main program declaration
30	int argc;	declaration for main argc argument
31	char* argv[];	declaration for main argv argument
32	#define	#define keyword
33	#undef	#undef keyword

Idn	Keyword	Description of use in C
34	if	if keyword
35	switch	switch keyword
36	case	case keyword
37	default	default keyword
38	break	break keyword
39	goto	goto keyword
40	for	for keyword
41	return	return keyword
42	DATAVAR	namelist structure for runtime data
43	DATAVAL	data for runtime data
44	END	marks end of runtime data
45	FIRST	variable which record first entry into function
46	IENTRY	entry point number variable
47	RETNUMB	alternate return variable
48	ELP	end-local-processing statement label
49	0	zero of type of INTEGER*2
50	0.0	zero of type of REAL*8
51	0	zero of type LOGICAL*2
52	'\0'	zero of type INTEGER*1
53	0L	zero of type INTEGER*4
54	0.0	zero of type REAL*4
55	0L	zero of type LOGICAL*4
56	0	zero of type LOGICAL*1
57	0.0	zero of type REAL*16
58	dpxzero	zero of type DOUBLE COMPLEX
59	cpxzero	zero of type COMPLEX
60	int	int keyword
61	NULL	NULL keyword
62	fiointu((char*)	initializes runtime data processing
63	fiornl	performs runtime data processing
65	union	union keyword
66	sizeof	sizeof keyword
119	INT2	i/o list designator for INTEGER*2
120	REAL8	i/o list designator for REAL*8
121	LOG2	i/o list designator for LOGICAL*2
122	BYTE	i/o list designator for INTEGER*1
123	INT4	i/o list designator for INTEGER*4
124	REAL4	i/o list designator for REAL*4
125	LOG4	i/o list designator for LOGICAL*4
126	LOG1	i/o list designator for LOGICAL*1
127	DCMPLX	i/o list designator for DOUBLE COMPLEX
128	STRG	i/o list designator for CHARACTER
129	DO	/o list designator for implied DO
130	CSTR	i/o list designator for constant string
131	\7%7d	printf for free-form INTEGER*2
132	\25%25.15E	printf for free-form REAL*8
133	\2%2c	printf for free-form LOGICAL*2
134	\3%3d	printf for free-form INTEGER*1
135	\12%12ld	printf for free-form INTEGER*4
136	\16%16.6E	printf for free-form REAL*4
137	\2%2c	printf for free-form LOGICAL*4
138	\3%3d	printf for free-form LOGICAL*1
139	\25%25.15LE	printf for free-form REAL*16
140	\53(%25.15E,%25.15E)	printf for free-form DOUBLE COMPLEX
141	\35(%16.6E,%16.6E)	printf for free-form COMPLEX
142	\0%s	printf for free-form CHARACTER
150	CMPLX	i/o list designator for COMPLEX
151	_Inline	inline function designator
152	#include "fortran.h"	include request for fortran.h
154	ftnblkd	name of BLOCK DATA subprogram
163	REAL8	i/o list designator for REAL*8

Idn	Keyword	Description of use in C
164	BD_	COMMON data module prefix
165	VOID	type of initialized structures array
166	#define LPROTOTYPE	platform type designator
169	volatile	volatile keyword
170	" = { NULL, 0 }"	initialization string for descriptors
171	"Z "	replacement characters for $ and -
211	ftnstruc	structure initializations of set array
212	ftnfdata	structure initializations data
213	ftnrecrd	initialized structures array
214	fiostrdi(ftnstruc,	structure data initialization function call ftnfdata,ftnrecrd);
259	"string"	type label for string descriptors
260	"n"	element name of descriptor string length
261	"a"	element name of descriptor string address
270	"TRUE"	symbol for logical TRUE constant value
271	"FALSE"	symbol for logical FALSE constant value
272	"Ktrue"	symbol for logical TRUE constant variable
273	"Kfalse"	symbol for logical FALSE constant variable

The pattern strings transform simple strings generated into alternate forms. The simple string is represented by a percent (%) sign within the pattern string. The table below lists the available pattern strings.

Idn	Keyword	Description of use in C
64	#include "%"	pattern string for writing includes
153	#pragma pfc("%")	pattern string for writing pragmas
155	void ftnblkd() {	name of COMMON initializations function
156	%1();	pattern string for COMMON initialization calls
157	}	end of COMMON initialization function
172	"%"	pattern string for external functions
174	"P%"	pattern string for argument surrogates
262	"T%"	pattern string for local COMMON variable
263	"C%"	pattern string for COMMON structure
264	"X%"	pattern string for COMMON external
265	"%.h"	pattern string for translated include names
267	"#endif /*ICF_% */"	pattern string for include #endif
268	"#ifndef ICF_%"	pattern string for include #ifndef
269	"#define ICF_%"	pattern string for include #define

The KEYWORD statement in the configuration file begins with the command KEYWORD as a single line. It is followed by a series of lines each containing a keyword string in quotes or a simple keyword identifier followed by the substitute string to be used in quotes. The end of the replacement pairs is indicated by the word END on a single line.

The following subsections give several examples in which the look of the translation of the following code is altered using a KEYWORDS statement.

```
SUBROUTINE TEST4
INTEGER*4 hd_hedsiz      ! /* Length of header */
INTEGER*4 hd_numtab      ! /* No. of tables in file */
INTEGER*4 hd_filrev      ! /* Model file revision date */
INTEGER*4 hd_sysprim     ! /* No. of system primitive types */
COMMON/MODELHEAD/hd_hedsiz,hd_numtab,hd_filrev,hd_sysprim
CALL TEST5(hd_numtab,hd_filrev)
hd_hedsiz = hd_numtab - hd_filrev
WRITE(1) hd_hedsiz,hd_numtab,hd_filrev,hd_sysprim
RETURN
END
SUBROUTINE TEST5(num,rev)
```

```
      INTEGER*4 num,rev
      num = num * rev + 2
      RETURN
      END
```

The default translation of this code is as follows:

```
void test4(heading,P1)
char *heading;
int P1;
{
extern void test5();
extern char Xmodelhead[];
typedef struct {
    long hd_hedsiz;               /* Length of header*/
    long hd_numtab;               /* No. of tables in file*/
    long hd_filrev;               /* Model file revision date*/
    long hd_sysprim;              /* No. of system primitive types*/
} Cmodelhead;
auto Cmodelhead *Tmodelhead = (Cmodelhead*) Xmodelhead;
    test5(&Tmodelhead->hd_numtab,&Tmodelhead->hd_filrev);
    Tmodelhead->hd_hedsiz = Tmodelhead->hd_numtab-Tmodelhead->hd_filrev;
    WRITE(1,LISTIO,STRG,heading,10,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_hedsiz,INT4,Tmodelhead->hd_numtab,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_filrev,INT4,Tmodelhead->hd_sysprim,0);
    return;
}
void test5(num,rev)
long *num,*rev;
{
    *num = *num**rev+2L;
    return;
}
char Xmodelhead[16];
```

3.5.1 Simple Keyword Replacement

To begin the detailed discussion of keyword replacement, let us assume that the user prefers FORTRAN looking variable type names. He has made the following additions to the fortran.h file

```
#define INTEGER4   long
#define CHARACTER  char
```

and wishes to have the corresponding changes made in his PROMULA output. The following `demo.cnf` file will do this.

```
KEYWORDS
6    "INTEGER4"
13   "CHARACTER"
END
```

When `test4` is translated using the `Rdemo` command line switch, the following translation will be produced.

```
void test4(heading,P1)
CHARACTER *heading;
int P1;
{
extern void test5();
extern char Xmodelhead[];
typedef struct {
    INTEGER4 hd_hedsiz;           /* Length of header*/
    INTEGER4 hd_numtab;           /* No. of tables in file*/
    INTEGER4 hd_filrev;           /* Model file revision date*/
    INTEGER4 hd_sysprim;          /* No. of system primitive types*/
```

```
} Cmodelhead;
auto Cmodelhead *Tmodelhead = (Cmodelhead*) Xmodelhead;
    test5(&Tmodelhead->hd_numtab,&Tmodelhead->hd_filrev);
    Tmodelhead->hd_hedsiz = Tmodelhead->hd_numtab-Tmodelhead->hd_filrev;
    WRITE(1,LISTIO,STRG,heading,10,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_hedsiz,INT4,Tmodelhead->hd_numtab,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_filrev,INT4,Tmodelhead->hd_sysprim,0);
    return;
}
void test5(num,rev)
INTEGER4 *num,*rev;
{
    *num = *num**rev+2L;
    return;
}
char Xmodelhead[16];
```

Notice that there are several different `char` occurrences within the output. Some pertain directly to the CHARACTER type while others are used for various memory allocation specifications. Keyword 13 is the one for the CHARACTER type. Alternatively, there is only one use of `long` in the keyword list; therefore, the following alternative `demo.cnf` file would produce the same result.

```
KEYWORDS
"long" "INTEGER4"
13     "CHARACTER"
END
```

3.5.2 Pattern Strings for COMMON blocks

In the default translation COMMON blocks generate three different symbols — distinguished by the initial first letter. The `X` indicates the global storage area, the `C` indicates the structure definition, and the `T` indicates local pointer. The pattern strings used to produce this convention are `"X%"`, `"C%"`, and `"T%"`. Suppose instead that you wish to label the global storage area with the suffix `_area`, to label the structure definition with the prefix `struct_`, and to have the local pointer simply use the COMMON name. The following KEYWORDS statement would make this change.

```
KEYWORDS
"X%"   "%_area"
"C%"   "struct_%"
"T%"   "%"
END
```

The following translation is produced.

```
void test4(heading,P1)
char *heading;
int P1;
{
extern void test5();
extern char modelhead_area[];
typedef struct {
    long hd_hedsiz;               /* Length of header*/
    long hd_numtab;               /* No. of tables in file*/
    long hd_filrev;               /* Model file revision date*/
    long hd_sysprim;              /* No. of system primitive types*/
} struct_modelhead;
auto struct_modelhead *modelhead = (struct_modelhead*) modelhead_area;
    test5(&modelhead->hd_numtab,&modelhead->hd_filrev);
    modelhead->hd_hedsiz = modelhead->hd_numtab-modelhead->hd_filrev;
    WRITE(1,LISTIO,STRG,heading,10,0);
    WRITE(1,LISTIO,INT4,modelhead->hd_hedsiz,INT4,modelhead->hd_numtab,0);
```

```
    WRITE(1,LISTIO,INT4,modelhead->hd_filrev,INT4,modelhead->hd_sysprim,0);
    return;
}
void test5(num,rev)
long *num,*rev;
{
    *num = *num**rev+2L;
    return;
}
char modelhead_area[16];
```

Of course the KEYWORDS of this section can be combined with those of the last to produce the following `demo.cnf` file.

```
KEYWORDS
"long" "INTEGER4"
13     "CHARACTER"
"X%"   "%_area"
"C%"   "struct_%"
"T%"   "%"
END
```

This configuration then produces the following translation.

```
void test4(heading,P1)
CHARACTER *heading;
int P1;
{
extern void test5();
extern char modelhead_area[];
typedef struct {
    INTEGER4 hd_hedsiz;           /* Length of header*/
    INTEGER4 hd_numtab;           /* No. of tables in file*/
    INTEGER4 hd_filrev;           /* Model file revision date*/
    INTEGER4 hd_sysprim;          /* No. of system primitive types*/
} struct_modelhead;
auto struct_modelhead *modelhead = (struct_modelhead*) modelhead_area;
    test5(&modelhead->hd_numtab,&modelhead->hd_filrev);
    modelhead->hd_hedsiz = modelhead->hd_numtab-modelhead->hd_filrev;
    WRITE(1,LISTIO,STRG,heading,10,0);
    WRITE(1,LISTIO,INT4,modelhead->hd_hedsiz,INT4,modelhead->hd_numtab,0);
    WRITE(1,LISTIO,INT4,modelhead->hd_filrev,INT4,modelhead->hd_sysprim,0);
    return;
}
void test5(num,rev)
INTEGER4 *num,*rev;
{
    *num = *num**rev+2L;
    return;
}
char modelhead_area[16];
```

3.5.3 Pattern String for External Functions

Many UNIX FORTRAN compilers — AIX and SUN OS for example — append an underscore to external symbols. This effect can be achieved via PROMULA by changing pattern string 172. The following configuration file

```
KEYWORDS
172 "%_"
END
```

produces the following translation. Notice the underscores appended to the function symbols.

```
void test4_(heading,P1)
char *heading;
int P1;
{
extern void test5_();
extern char Xmodelhead[];
typedef struct {
    long hd_hedsiz;               /* Length of header*/
    long hd_numtab;               /* No. of tables in file*/
    long hd_filrev;               /* Model file revision date*/
    long hd_sysprim;              /* No. of system primitive types*/
} Cmodelhead;
auto Cmodelhead *Tmodelhead = (Cmodelhead*) Xmodelhead;
    test5_(&Tmodelhead->hd_numtab,&Tmodelhead->hd_filrev);
    Tmodelhead->hd_hedsiz = Tmodelhead->hd_numtab-Tmodelhead->hd_filrev;
    WRITE(1,LISTIO,STRG,heading,10,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_hedsiz,INT4,Tmodelhead->hd_numtab,0);
    WRITE(1,LISTIO,INT4,Tmodelhead->hd_filrev,INT4,Tmodelhead->hd_sysprim,0);
    return;
}
void test5_(num,rev)
long *num,*rev;
{
    *num = *num**rev+2L;
    return;
}
char Xmodelhead[16];
```

3.5.4 Pattern String for Subprogram Surrogates

As is discussed under the Y4 command line switch a typical FORTRAN optimization involves making local surrogates for subprogram parameters. Using Y4 with the example used above produces the following translation (only last subprogram shown).

```
void test5(Pnum,Prev)
long *Pnum,*Prev;
{
auto long num = *Pnum;
auto long rev = *Prev;
    num = num*rev+2L;
    goto ELP;
ELP:
    *Pnum = num;
}
```

The manner in which the local surrogates are formed can easily be changed. Consider the following configuration file

```
KEYWORDS
"P%"    "Local_%_copy"
END
```

which will change the look of the translation considerably.

```
void test5(Local_num_copy,Local_rev_copy)
long *Local_num_copy,*Local_rev_copy;
{
auto long num = *Local_num_copy;
auto long rev = *Local_rev_copy;
    num = num*rev+2L;
    goto ELP;
ELP:
    *Local_num_copy = num;
```

```
}
```

3.5.5 Pattern Strings for VAX Descriptors

Users of VAX FORTRAN often want to treat characters as descriptors. The CHv flag gives this basic treatment. Consider the following simple code.

```
      SUBROUTINE DEMO(I,J)
      CHARACTER*12 NAME
      CHARACTER*4 INITIAL
      I = INDEX(NAME,INITIAL)
      J = JINDEX(NAME,INITIAL)
      RETURN
      END
```

The output of the code above using the CHv command line switch — though we will put the switch into a configuration file as discussed earlier in this chapter — is below.

```
void demo(i,j)
long *i,*j;
{
extern long jindex();
static char name[12],initial[4];
static string T1 = { NULL, 0 };
static string T2 = { NULL, 0 };
    *i = fifindex(name,12,initial,4);
    T1.a = name; T1.n = 12;
    T2.a = initial; T2.n = 4;
    *j = jindex(&T1,&T2);
    return;
}
```

All the machinery needed for "real" VAX descriptors is here; however, the following configuration file is needed.

```
SWITCHES CHv
KEYWORDS
"string"          "DESCRIPTOR"
"n"               "Dleng"
"a"               "Dptr"
"ftnsallo"        "ftndallo"
" = { NULL, 0 }"  " = { 0, 0, 0, NULL }"
END
```

This produces the following translation.

```
void demo(i,j)
long *i,*j;
{
extern long jindex();
static char name[12],initial[4];
static DESCRIPTOR T1 = { 0, 0, 0, NULL };
static DESCRIPTOR T2 = { 0, 0, 0, NULL };
    *i = fifindex(name,12,initial,4);
    T1.Dptr = name; T1.Dleng = 12;
    T2.Dptr = initial; T2.Dleng = 4;
    *j = jindex(&T1,&T2);
    return;
}
```

3.6 The Configuration PRAGMA Statement

Sotfware developers have long faced the problem of nonportability with FORTRAN. As a result, many have written FORTRAN preprocessors that aid in making FORTRAN programs more portable. A major feature of C that makes it portable is its ability to deal with conditional compilation. It is very common for FORTRAN preprocessors to have such a capability as well. Consider the following simple FORTRAN code which uses a conditional compilation notation.

```
          SUBROUTINE TEST6
          CALL RENAME_F ('modxx.lio','model.lio')
*#IF UNIX
*?        CALL RENAME_F ('modxx.lio','model.lio')
*#ENDIF
          CALL RENAME_F ('modxx.lio','model.lio')
          STOP
          END
```

This particular notation uses three particular comment strings:

`*#IF`	introduces a conditional expression
`*?`	introduces a statement within a conditional block
`*#ENDIF`	closes a conditional block

Though the notation varies from site to site, the above is typical. Since these conventions are based on C, we need to have some mechanism for describing them to PROMULA.

To see the problem, the following is the default translation of the above.

```
void test6()
{
extern void rename_f();
    rename_f("modxx.lio","model.lio",9,9);
/*
#IF UNIX
?       CALL RENAME_F ('modxx.lio','model.lio')
#ENDIF
*/
    rename_f("modxx.lio","model.lio",9,9);
    STOP(NULL);
}
```

The translator completely ignores the true meaning of the condition block. The whole thing is treated as a comment. The fact that the `*?` should be ignored when going to C is easily established using the YD*? command line switch. This switch is described in the previous chapter. It gives the following translation.

```
void test6()
{
extern void rename_f();
    rename_f("modxx.lio","model.lio",9,9);
/*
#IF UNIX
*/
    rename_f("modxx.lio","model.lio",9,9);
/*
#ENDIF
*/
    rename_f("modxx.lio","model.lio",9,9);
    STOP(NULL);
}
```

Now we have gone too far the other way. The conditional statement will always be executed. We must tell PROMULA to translate the `#IF` and `#ENDIF` into `#if` and `#endif`. The following configuration file does this.

```
SWITCHES  Yd*?
PRAGMA "#" "IF" "#ifdef%" "ENDIF" "#endif"
```

Syntax:

```
PRAGMA "c" "s1" "t1" ... "sn" "tn"
```

Where:

- `c` is the comment character introducing a special comment
- `si` is a keyword in the source language
- `ti` is the corresponding pattern string in the target language where the `%` represents anything on the comment following the keyword.

In the PRAGMA statement above then, conditional statements begin with a `#.` The `IF` conditional is translated into `#ifdef` conditional and "`ENDIF` whatever" is translated into `#endif`.

The resultant translation of the code above is as follows.

```
void test6()
{
extern void rename_f();
    rename_f("modxx.lio","model.lio",9,9);
#ifdef UNIX
    rename_f("modxx.lio","model.lio",9,9);
#endif
    rename_f("modxx.lio","model.lio",9,9);
    STOP(NULL);
}
```

3.7 The Configuration $ Statement

The configuration $ statement simply specifies the replacement character for the dollar sign.

Syntax:

```
$c
```

Where:

- `c` is the replacement character.

By default all `$` symbols in the source code are replaced by `Z`. The default translation of the following code

```
FUNCTION HELLO$(I$,J$)
DIMENSION WORLD$(10,20)
HELLO$ = WORD(I$,J$)
RETURN
END
```

is as follows.

```
float helloZ(iZ,jZ)
long *iZ,*jZ;
{
extern float word();
static float helloZ;
    helloZ = word(iZ,jZ);
    return helloZ;
```

```
}
```

Using a configuration statement of $S would give the following.

```
float helloS(iS,jS)
long *iS,*jS;
{
extern float word();
static float helloS;
    helloS = word(iS,jS);
    return helloS;
}
```

Alternatively, if $is to be left alone, use $$.

```
float hello$(i$,j$)
long *i$,*j$;
{
extern float word();
static float hello$;
    hello$ = word(i$,j$);
    return hello$;
}
```

4. THE CONFIGURATION FUNCTION PROTOTYPES

As has been discussed in several other places, FORTRAN passes all function arguments by name — i.e., it passes the address of a parameter to a subprogram as opposed to its value. C, on the other hand, allows argument values to be passed directly as well as allowing the passing of argument addresses. In instances where the value of the argument is scalar and where its value is not being changed by the subprogram, passing the value of that argument is far more efficient than passing its address. Whenever possible, the translation from FORTRAN to C should use call-by-value. The problem is that it is not always possible tell whether call-by-value is possible simply from the source code. Some other device is needed to tell the translator which arguments can be passed by value.

Another problem with FORTRAN is that it is weak typing. By this is meant that it is sometimes valid to pass data of differing binary types via the same subprogram argument. The translator needs to know when this is valid; and if it is not valid, it needs to know which binary type is the expected one. Many perfectly valid FORTRAN programs fail either at compilation time or execution time because of differing typing conventions and differing internal representations. There is no general solution to the translation of "weak-typing" FORTRAN programs. The translator must be told what to do. Some device is needed to describe subprogram arguments to the translator.

A similar problem has faced C programmers as well. Consequently, the new ANSI C has introduced the notion of a "function prototype" which describes the arguments of functions in terms of their binary type and in terms of their pointer status. The conventions developed there are exactly those needed by the translator, although they need to be extended slightly to deal with virtual variables, multiple forms, and "external name clash".

The C prototype system and its extensions to PROMULA are discussed in this section. It is strongly recommended that you take the time to develop a set of prototypes for the subprograms within a FORTRAN program prior to translation, if that program is suspected of playing any games with internal representations.

It should be pointed out that the storage of prototype information read from a configuration file has been carefully optimized, which means that there are minimal performance penalties for using large prototype files.

4.1 Function Prototype Syntax

In C, a "function prototype" declaration defines the name, return type, and storage class of a function. In addition, it can define the types of some or all of the arguments for that function. The prototype declaration has the same format as the function declaration, except that it is terminated by a semicolon, and the argument identifiers are optional. If used, argument identifiers have scope only within the prototype declaration and serve only as place holders — i.e., int a,b,c; is the same as int,int,int.

The syntax used for prototypes by PROMULA is similar to that used by C; however, it is unfortunately not the same. The reason is that additional information must be supplied to the translator since additional problems are introduced by the weak-typing conventions of FORTRAN.

The syntax of the PROMULA prototype definition is as follows:

```
[fname] type name(type[c],type[c][,...])
      [,type name(type[c],type[c][,...])[,...]
```

Where:

`fname` is the name of the subprogram used in the FORTRAN code.

`type` is one of the following C binary type specifiers:

`void`

```
short
double
unsigned short
char
long
float
unsigned long
dcomplex
complex
string
```

`name` is any valid identifier to be used in the C output

`c` is one of the following special characters:

`*`	indicates a memory pointer to the indicated type
`+`	indicates a virtual pointer to the indicated type
`!`	indicates that a value conversion should be made to the indicated type

Only a single prototype definition may occur on each record and the notation below in PROMULA

```
type name()
```

means that the function has no arguments. It does not mean that the function has some unspecified number of unspecified arguments.

The type specifiers have their traditional C interpretation. The complex and dcomplex types refer to single-precision complex and double-precision complex respectively. The string type refers to the FORTRAN style character string which was discussed extensively in the section on the CHARACTER type. Remember that a string argument actually consists of a pointer and a length specification. Depending upon the character translation scheme being used, these two may not even appear together in the actual C function declaration.

4.2 Value Parameters

If no c specifier follows the type specification, a value parameter is being defined. When a reference to this argument is processed, the value of the argument is passed and not its address. If the actual argument does not have the proper type, then an error occurs. The only exception to this occurs when processing numeric constants. A constant with a decimal point or exponent specified may be used as either a float or double value parameter, and a numeric constant without a decimal indication with a value in the range -32767 to + 32767 may be used in either a long or short environment. In a long context an "L" is appended. The following simple example shows this.

```
void alpha(float,double,short,long)
```

```
      SUBROUTINE DEMO
      CALL ALPHA(1.234,10E5,32,32)
      RETURN
      END
      SUBROUTINE ALPHA(FLT,DBL,ISHT,LNG)
      REAL*8 DBL
      INTEGER*2 ISHT
      WRITE(*,*) FLT,DBL,ISHT,LNG
      RETURN
      END
```

The prototype above used in conjunction with the FORTRAN code above produces the following output.

```
void demo()
{
extern void alpha();
```

```
    alpha(1.24,1000000.0,32,32L);
    return;
}
void alpha(flt,dbl,isht,lng)
int isht;
double dbl;
long lng;
float flt;
{
    WRITE(OUTPUT,LISTIO,REAL4,flt,REAL8,dbl,INT2,isht,INT4,lng,0);
    return;
}
```

In this output, the 32 being passed via the long parameter is shown as a long constant. The following is the same translation as the above, but without the prototype supplied.

```
void demo()
{
extern void alpha();
static float K1 = 1.234;
static long K3 = 32;
static double T2;
    T2 = 1000000.0;
    alpha(&K1,&T2,&K3,&K3);
    return;
}
void alpha(flt,dbl,isht,lng)
int *isht;
double *dbl;
long *lng;
float *flt;
{
    WRITE(OUTPUT,LISTIO,REAL4,*flt,REAL8,*dbl,INT2,*isht,INT4,*lng,0);
    return;
}
```

Notice that PROTOTYPE definitions effect both the translation of references to subprograms and the definitions of subprograms. In this sense, PROMULA is very different than C itself. Clearly the version driven by the prototype is the better version — both from the standpoint of readability and from the standpoint of efficiency.

4.3 External Name Clash

A problem that pervades C, in particular, is the external name clash problem. One tends to use many different libraries with C. It is not at all unusual to have the same names used by different libraries. As an example, we recently worked with a FORTRAN program which had two generalized I/O routines named fread and fwrite, which stood for FORTRAN READ and FORTRAN WRITE. Since these are the names of the standard C I/O functions they needed to be changed. The prototype system in PROMULA allows this to be handled very easily. Consider the following example translated using the prototype definition shown.

```
fread void ftread(short,long*,short)
fwrite void ftwrite(short,long*,short)

      SUBROUTINE DEMO
      INTEGER A,B(10),C(20)
      CALL FREAD(1,A,1)
      CALL FWRITE(2,A,1)
      CALL FREAD(1,B,10)
      CALL FWRITE(2,B,10)
      CALL FREAD(1,C,20)
      CALL FWRITE(2,C,20)
      RETURN
      END
```

In the prototype, the name as it appears in the FORTRAN comes first, followed by the standard prototype information. The decision to change these names is made entirely in the prototype file, no changes are needed in the actual FORTRAN. The C output looks as follows.

```
void demo()
{
extern void ftread();
extern void ftwrite();
static long a,b[10],c[20];
    ftread(1,&a,1);
    ftwrite(2,&a,1);
    ftread(1,b,10);
    ftwrite(2,b,10);
    ftread(1,c,20);
    ftwrite(2,c,20);
}
```

The old names have been completely replaced as a result of the prototype specification.

4.4 Multiple Forms

Another problem that comes up has to do with multiple forms. Here a single function in FORTRAN has been translated in different ways either because of some weak typing convention or because the function is to be used in both virtual and non-virtual mode. As an example, consider that you have a statistical analysis function which computes the mean and variance of a vector of values. You have translated it twice, once using virtual conventions and once using memory conventions. Let us first see how these two versions of the following utility can be produced.

```
      SUBROUTINE ANADAT(VAL,N,XBAR,VAR)
      DIMENSION VAL(N)
      XBAR=0.0
      VAR=0.0
      DO 10 J = 1,N
      XBAR = XBAR + VAL(J)
   10 CONTINUE
      XBAR = XBAR/N
      DO 15 J = 1,N
      S = VAL(J) - XBAR
      VAR = VAR + S*S
   15 CONTINUE
      VAR = VAR/(N-1)
      RETURN
      END
```

Translating it with the following prototype produces the memory version shown below.

```
anadat void manadat(float*,short,double*,double*)

void manadat(val,n,xbar,var)
int n;
double *xbar,*var;
float val[];
{
static long j;
static float s;
    *xbar = 0.0;
    *var = 0.0;
    for(j=0L; j<n; j++) {
        *xbar = *xbar+val[j];
    }
    *xbar = *xbar/(double)n;
```

```
    for(j=0L; j<n; j++) {
        s = val[j]-*xbar;
        *var = *var+s*s;
    }
    *var = *var/(double)(n-1);
    return;
}
```

Notice that the name of the function has been changed to agree with the prototype name. Now translating the identical FORTRAN code with the following prototype produces the virtual code shown.

```
anadat void vanadat(float+,short,double*,double*)

void manadat(val,n,xbar,var)
int n;
double *xbar,*var;
long val;
{
static long j;
static float s;
    *xbar = 0.0;
    *var = 0.0;
    for(j=0L; j<n; j++) {
        *xbar = *xbar+*(float*)vmsuse(val+j*4);
    }
    *xbar = *xbar/(double)n;
    for(j=0L; j<n; j++) {
        s = *(float*)vmsuse(val+j*4)-*xbar;
        *var = *var+s*s;
    }
    *var = *var/(double)(n-1);
    return;
}
```

This code treats `val` as a long virtual address and not as an array. All references to val are made through the vmsuse function. Again the name of the function has been changed via the prototype.

Now the following prototype and FORTRAN code can be translated. In the translation we will tell PROMULA that C is to be virtual, via an Sv100 command line switch.

```
anadat void vanadat(float+,short,double*,double*),
       void manadat(float*,short,double*,double*)

      SUBROUTINE TEST
      DIMENSION A(10),B(20),C(100)
      REAL*8 ABAR,AVAR,BBAR,BVAR,CBAR,CVAR
      CALL ANADAT(A,10,ABAR,AVAR)
      CALL ANADAT(B,20,BBAR,BVAR)
      CALL ANADAT(C,100,CBAR,CVAR)
      RETURN
      END
```

In this function `ANADAT` will be translated to use `manadat` when the vector is in memory and `vanadat` when the vector is virtual. The translation is as follows:

```
void test()
{
extern void vanadat();
extern void manadat();
static double abar,avar,bbar,bvar,cbar,cvar;
static float a[10],b[20];
static long c=32;
```

```
        manadat(a,10,&abar,&avar);
        manadat(b,20,&bbar,&bvar);
        vanadat(c,100,&cbar,&cvar);
        return;
    }
```

This translation is exactly as desired.

4.5 Global Symbols and Prototypes

To interface with external subroutines, functions, and common data areas PROMULA must know the naming conventions and parameter types for each. This information is supplied via function prototypes, using either standard ANSI C notation or an extended notation which allows changing the name of the external in the target language. These prototypes are entered within separate prototype files which are read at runtime via the Rfilename command line switch. The notation and use of these files is described in detail in the earlier sections of this chapter.

An additional problem that comes up is that some of the runtime libraries have been implemented via FORTRAN and others via C. Some FORTRANs and/or some Cs append additional characters to each external symbol — be it a function or a subroutine. This requires that groups of global symbols, but not all, use a modified naming convention. These modifications are achieved via GLOBAL strings, which may be entered into prototype files.

The GLOBALS string consists of two characters only — the function or subroutine prefix and suffix characters. As an example, consider the following piece of a prototype file for a FORTRAN based system in which the FORTRAN compiler appends an underscore character.

```
GLOBALS " _"
void actday(long*,long*,long*,long*);
void valdt(long*,long*,long*,short*);
void fixdt(long*,long*,long*,long*,long*,long*,long*,short*);
void weeknd(long*,long*,long*,long*,long*,long*,long*,short*);
```

The actual public names are `actdat_`, `valdt_`, `fixdt_`, and `weeknd_`. The GLOBALS string preceding these tells PROMULA to make this change in the target C.

4.6 Renaming Identifiers Only

A final capability of the configuration file is to allow the user to enter simple identifier renaming requests. The following notation

```
oldname * newname
```

in a prototype input file will rename all occurrences of oldname with newname.

5. OVERVIEW OF RUNTIME LIBRARY

The PROMULA FORTRAN runtime library is a set of approximately 250 functions designed for use with the C output of PROMULA FORTRAN. It may also be used by those FORTRAN programmers who wish to program in C, but who do not wish to give up the input/output conventions, formatting controls, and intrinsic functions which they have grown used to.

This chapter is provided primarily for those user's of PROMULA FORTRAN who intend to maintain the C output independently of their FORTRAN originals. Initially, C codes using this library can be produced by producing C source code using PROMULA FORTRAN. Once in C, the programs may then be maintained by using these functions and their documentation.

The source code for all functions in this library is available. If there is one certainty, it is that no two FORTRANs behave in the same way, especially with regard to their runtime libraries. Thus, if your conventions differ from the ones used here, or if you require some specialized behavior, you may alter the library code. Alternatively, your version of FORTRAN may contain statements which require runtime support not included in this library. In this case, you can add the additional functions needed. For more discussion of this topic see the chapter in this manual on the implementation of nonstandard FORTRAN dialects.

Another use of the material in this and the next chapter is to optimize the runtime behavior of the C output for special platforms or special needs. Much of this customization can be done via the `fortran.h` file which is included with each C output from PROMULA FORTRAN. This file is discussed in this chapter as well.

5.1 Naming and Organization of Functions

The PROMULA FORTRAN runtime library functions are divided into six general groups. Associated with each group is a three letter prefix which is part of each function name. The groups are as follows:

Prefix	General Description
`cpx`	Performs single precision complex arithmetic
`dpx`	Performs double precision complex arithmetic
`fif`	Noncomplex intrinsic function
`fio`	Performs an input/output operation
`ftn`	Performs a general FORTRAN operation
`p77`	PRIME FORTRAN functions
`pdp`	PDP FORTRAN functions
`vms`	virtual memory management functions

The descriptions of the individual functions are given in alphabetic order in the following chapter. The remainder of this chapter gives a summary discussion of the six functions, a listing of the runtime error produced, a description of the `pfclib.h` header file which is included with every runtime function, and finally a description of the `fortran.h` header file which is included by `pfclib.h` and which is also included by each C output file produced by PROMULA FORTRAN.

The final file needed by every runtime function is the `platform.h` file. This file is used to achieve transportability of the C source code over a wide variety of platforms. This header file is also included by the shrouded source code of PROMULA FORTRAN itself. It is discussed in the chapter containing the installation instructions for PROMULA FORTRAN and its runtime library.

5.2 General FORTRAN Operations

This group of functions is named with the prefix ***ftn***. They include those operations needed to support the actual FORTRAN language conventions; character manipulation, 3-way branching, generalized DO loops, etc. Also included in this group are the interface functions used by the C and FORTRAN bias to simplify the translation of FORTRAN I/O statements.

5.3 Input/Output Operations

This group of functions is named with the prefix ***fio***. As has been discussed in other parts of this manual, the FORTRAN input/output statements have not been translated into standard C type input/output operations. Rather, they have been translated into function calls much like the calls that would be generated by an actual FORTRAN compiler.

Each FORTRAN statement is broken into a series of function calls. If the FORTRAN operation specifies any error operations, then an initial call is made to a function `fiostatus`, which establishes the error processing conventions to be used, and a final call is made to `fioerror` which generates any error branches requested. If no error processing is specified by the FORTRAN statement, then an error causes the program to exit.

5.3.1 Runtime Error Messages

The actual I/O errors generated along with the name of the function generating them are listed below. The identifier of each error is a mneumonic which specifies the function which generated it and its cause. The codes specified are those returned if the I/O statement requests an error status value.

Error	Code	Function	Description
ELUN_EOF	-1	fiolun	End of file encountered
EOPN_EOF	-1	fioopen	End of file encountered
ERBV_EOF	-1	fiorbiv	End of file encountered
ERTX_EOF	-1	fiortxt	End of file
EWTX_EOF	-1	fiowtxt	Write beyond end of file
ELUN_RDO	106	fiolun	Write to read only file
EINT_NAF	107	fiointu	No active file structure
ELUN_NAF	108	fiolun	No active file structure
EINT_TMF	109	fiointu	Too many files open
ELUN_TMF	110	fiolun	Too many files open
ENAM_TMF	111	fioname	Too many files open
ESIO_TMF	112	fiostio	Too many files open
ECLO_PCF	113	fioclose	Physical close failure
EOPN_POF	114	fioopen	Physical open failed
ECLO_POF	115	fioclose	Physical open failure
EBCK_FRT	116	fioback	At front of file
EBCK_DIR	117	fioback	Direct access file
ELUN_PWF	118	fiolun	Physical write failed
ERWV_PWF	119	fiorwbv	Physical write failure
EWBV_PWF	120	fiowbiv	Physical write failure
EWEF_PWF	121	fiowef	Physical write failure
ERDB_IFS	122	fiordb	Invalid format specification
ERDD_IFS	123	fiordd	Invalid format specification
ERDF_IFS	124	fiordf	Invalid format specification
ERDI_IFS	125	fiordi	Invalid format specification
ERDL_IFS	126	fiordl	Invalid format specification
ERDS_IFS	127	fiords	Invalid format specification
ERDT_IFS	128	fiordt	Invalid format specification
EWRB_IFS	129	fiowrb	Bad format specification
EWRS_IFS	130	fiowrs	Bad format specification
EWRT_IFS	131	fiowrt	Bad format specification
EWVL_IFS	132	fiowval	Bad format specification

ENXF_EFS	133	fionxtf	End of format string
ENXF_BTF	134	fionxtf	Bad T format
ENXF_BUS	135	fionxtf	Bad B business format string
ENXF_BBF	136	fionxtf	Bad BN,Z format
ENXF_WID	137	fionxtf	Missing width specification
ENXF_DEL	138	fionxtf	Missing terminating delimiter
ENXF_HOL	139	fionxtf	Bad Hollerith string
ERCK_BUF	140	fiorchk	Internal buffer exceeded
ERDX_MLP	141	fiordx	Missing left parenthesis
ERDX_COM	142	fiordx	Missing comma
ERDX_MRP	143	fiordx	Missing right parenthesis

Error	Code	Function	Description
ERDZ_MLP	144	fiordx	Missing left parenthesis
ERDZ_COM	145	fiordx	Missing comma
ERDZ_MRP	146	fiordx	Missing right parenthesis
ERNL_MNI	147	fiornl	Missing namelist identifier
ERNL_MVI	148	fiornl	Missing variable identifier
ERNL_UVI	149	fiornl	Undefined variable identifier
ERNL_SSV	150	fiornl	Subscripted scalar variable
ERNL_NNS	151	fiornl	Nonnumeric subscripts
ERNL_TMS	152	fiornl	Too many subscripts
ERNL_EQL	153	fiornl	Missing equals sign
ERNL_BSI	154	fiornl	Bad string input
ERNL_MLP	155	fiornl	Complex missing left pren
ERNL_COM	156	fiornl	Complex missing comma
ERNL_MRP	157	fiornl	Complex missing right pren
ESTD_NNC	158	fiostod	Nonnumeric character in field

5.4 Noncomplex Intrinsic Functions

This group of functions is named with the prefix ***fif***. Many of the noncomplex intrinsic functions are part of the C "standard" C libraries. Insofar as the implementation of the noncomplex intrinsic functions is concerned, the following functions are assumed to be part of the library supplied with your C compiler.

Name	Prototype	Description of computation
acos	double acos(double)	arccosine
asin	double asin(double)	arcsin
atan	double atan(double)	arctangent
atan2	double atan2(double,double)	arctangent of y/x
cos	double cos(double)	cosine
cosh	double cosh(double)	hyperbolic cosine
exp	double exp(double)	exponential
fabs	double fabs(double)	absolute value
floor	double floor(double)	largest integer less than
log	double log(double)	natural logarithm
log10	double log10(double)	base 10 logarithm
sin	double sin(double)	sine
sinh	double sinh(double)	hyperbolic sine
sqrt	double sqrt(double)	square root
tan	double tan(double)	tangent
tanh	double tan(double)	hyperbolic tangent

The functions above are referenced directly by PROMULA FORTRAN in its translations of the intrinsic functions. In addition, the following functions are used internally in the implementation of the remaining functions.

Name	Prototype	Description of computation
modf	double modf(double,double*)	Gets components of a value
pow	double pow(double,double)	Raises value to a power

5.5 Virtual Memory System

This group of functions is named with the prefix ***vms***. The virtual memory system serves two very different, yet complementary purposes. The first is to allow programs with very large data needs to be implemented on systems with limited memory. The second is to make the information within programs available to the PROMULA Application Development System and to other programs and tools constructed using the Promula system. This topic is discussed extensively in the chapter on the PROMULA interface in the PROMULA FORTRAN Compiler Manual. To achieve these two goals the virtual file manager uses the same data structure as the PROMULA Application Development System.

The hardest problem which must be faced by a virtual memory system is determining whether or not a current block is in memory or is on the file. The approach taken by this implementation uses a block status byte. The total virtual memory space is divided into n evenly sized blocks, say N of them. A vector N bytes long is then reserved in memory. For any given block, if the block status is zero, then the block is not in memory. If the status byte is nonzero, then it specifies which memory block contains it. The advantage of this approach is that it is very fast. The disadvantages are as follows:

1. The overall size of the virtual memory space needed must be known in advance. This is no problem for this implementation because either a fixed PROMULA file is used (with known size when it is opened) or the translator has calculated this value during the translation phase.

2. The maximum memory space that can be used is 254 times the memory block size, which is 1024 bytes in this implementation. The current maximum is then 254K which is clearly ample given a 640K overall memory limit on the typical MS-DOS PC.

3. The maximum file size that can be accommodated is limited by the maximum length of the block status vector. Each K of memory space allows for a megabyte of virtual space, given the 1024 blocksize being used here. So, this is no problem. A minor problem faced by this algorithm is that the criteria for writing a block to the disk is based on its use pattern; consequently, a block high in the file might be written before some blocks below it, thus creating potential 'holes' in the file. Though such holes are acceptable to many operating systems, we have decided to avoid them here. The virtual algorithm must, therefore, remember whether a given block has been placed on the disk because it has valid information or simply to fill a hole.

5.5.1 The Virtual Memory Management Algorithm

The virtual memory management system is controlled via three basic storage elements:

2 The block status vector
3 The memory block address vector
4 The virtual memory block

The block status vector is the key to this algorithm. There must be one entry for each block of memory. To keep this vector as memory efficient as possible its entries can be stored as unsigned bytes. The particular values in this vector are as follows.

Code	Meaning

0	This block is not in memory and has no valid data on disk. See discussion above on holes.
1	This block is not now in memory, but has data on disk 2-255. This block is in memory and is stored in block i - 2

Given the block status vector, the current status of any memory block can be determined via a single memory access — no searching for the block is required. Note that the block status vector is dynamically allocated when the virtual memory system is initialized.

Assuming first that the desired block is specified as being in memory by the block status vector, then the memory block address vector contains a pointer to the actual virtual memory block. Since there is an absolute maximum of 254 memory blocks, the maximum number of entries in the memory block address vector is 254; therefore, this vector is statically allocated.

The virtual memory blocks are dynamically allocated until there are either 254 of them or until available dynamic memory is exhausted. Each block is defined via the `vmsbtyp` structure defined as follows:

```
#define VMSBSIZ   1024

typedef struct {
    int vbdel;
    int vblru;
    int vbmru;
    long vbadr;
    unsigned char vbdat[VBLKSIZ];
} vmsbtyp;
```

The members of the structure are defined as follows:

Name	Description of use
`vbdel`	A flag indicating whether or not the data in this block has been changed.
`vblru`	The number of that block which immediately precedes this block in the least-recently used chain. A zero in this member means that this block is the least-recently used block.
`vbmru`	number of that record which is immediately more-recently used than the current block or zero if this block is the most-recently used block.
`vbdat`	the actual data contained within this block.

As can be seen from the above structure, the virtual memory system keeps track of the order of reference of the blocks. When a block must be moved into memory and there are no empty memory blocks, the least recently used memory block is flushed.

5.5.2 Virtual Memory Global Variables

Internally, the virtual memory system functions are controlled via a set of statically allocated global variables. These are as follows.

Variable	Description of use

`vmsbadr`	The addresses in memory of virtual blocks. Note that there are a maximum of 254 virtual memory blocks; therefore, this array is statically allocated.
`vmsblock`	A pointer to the block status vector as described above.
`vmsfd`	The file descriptor for the virtual file.
`vmslru`	Number of the least-recently used block
`vmsnblk`	The number of blocks on the virtual file.
`vmsnwr`	The number of bytes currently written to the virtual file. This value is needed for the whole avoidance logic.

5.6 Single Precision Complex Arithmetic

This group of functions is named with the prefix ***cpx***. Single precision complex arithmetic is handled entirely via function references since C contains no complex arithmetic of its own. The functions are straightforward as can be seen in the following function descriptions. The basic structure used to implement the complex data type is as follows:

```
typedef struct {
    float cr;           /* The real part of the value */
    float ci;           /* The imaginary part of the number */
} complex
```

5.7 Double Precision Complex Arithmetic

Double precision complex arithmetic is handled entirely via function references since C contains no complex arithmetic of its own. The functions all begin with the prefix ***dpx*** and are straightforward as can be seen in the following function descriptions. The basic structure used to implement the double complex data type is as follows:

```
typedef struct {
    double cr;           /* The real part of the value */
    double ci;           /* The imaginary part of the number */
} dcomplex
```

6. RUNTIME LIBRARY FUNCTION DESCRIPTIONS

The following sections contain the descriptions for the functions in the PROMULA FORTRAN Runtime Library.

6.1 CPXABS: Compute the Short Complex Absolute Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

float cpxabs(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the value of the short complex absolute value. The result is single precision.

Return value:

The single precision result.

See also: None

6.2 CPXADD: Short Complex Addition

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

float cpxadd(a,b)
complex a                   The left-hand value
complex b                   The right-hand value
```

Description:

Adds two short complex numbers to form a third.

Return value:

The complex result of the addition.

See also: None

6.3 CPXCJG: Compute the Short Complex Conjugate

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxcjg(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the complex conjugate of a short complex value. By definition:

```
c = conj(a) = a.cr - i * a.ci
```

Return value:

The complex result.

See also: None

6.4 CPXCMP: Short Complex Comparison

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

int cpxcmp(a,b)
complex a                   The left-hand value
complex b                   The right-hand value
```

Description:

Compares two short complex numbers.

Return value:

A zero if the numbers are the same, else a one.

See also: None

6.5 CPXCOS: Compute the Short Complex Cosine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxcos(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the value of the short complex cosine of a short complex number. By definition:

$$\cos(a) = \frac{e^{(i * a)} + e^{(-i * a)}}{2}$$

Return value:

The short complex result.

See also:

float fifsncs; Compute single precision sin/cosine

6.6 CPXCPX: Convert Two Floats to Short Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxcpx(d1,d2)
double d1                   Contains the real value
double d2                   Contains the imaginary value
```

Description:

Forms a short complex number whose real and imaginary parts are specified values.

Return value:

The short complex result.

See also: None

6.7 CPXDBL: Convert Double Precision to Short Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxdbl(dbl)
double dbl                  Contains the value to be transformed
```

Description:

Forms a short complex number whose real part is a specified value and whose imaginary part is zero.

Return value:

The short complex result.

See also: None

6.8 CPXDIV: Short Complex Division

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxdiv(a,b)
complex a                   The numerator
complex b                   The denominator
```

Description:

Divides two short complex numbers to form a third.

Return value:

The short complex result of the division.

See also: None

6.9 CPXDPX: Convert Double Complex to Short Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double dr                The real part of the value
   double di                The imaginary part of the number
} dcomplex

float cpxdpx(dbl)
dcomplex dbl                Contains the value to be transformed
```

Description:

Forms a short complex number whose real and imaginary parts correspond to a double complex number.

Return value:

The short complex result.

See also: None

6.10 CPXEXP: Short Complex Exponential

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxexp(a)
complex  a                  The value to be transformed
```

Description:

Computes the short complex exponential of a short complex value.

Return value:

The short complex exponential.

See also: None

6.11 CPXIMA: Compute the Imaginary Part of a Short Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex
```

```
float cpxima(a)
complex a                    Contains the value to be transformed
```

Description:

Computes the imaginary part of a short complex value. The result is single precision.

Return value:

The single precision result.

See also: None

6.12 CPXLOG: Short Complex Natural Logarithm

Synopsis:

```
#include "fortran.h"         PROMULA FORTRAN function declarations

typedef struct {
   float cr                  The real part of the value
   float ci                  The imaginary part of the number
} complex

complex cpxlog(a)
complex a                    The value to be transformed
```

Description:

Computes the short complex natural logarithm of a short complex value.

Return value:

The complex natural logarithm.

See also: None

6.13 CPXLOG10: Short Complex Base 10 Logarithm

Synopsis:

```
#include "fortran.h"         PROMULA FORTRAN function declarations

typedef struct {
   float cr                  The real part of the value
   float ci                  The imaginary part of the number
} complex

complex cpxlog10(a)
complex a                    The value to be transformed
```

Description:

Computes the short complex base 10 logarithm of a short complex value.

Return value:

The complex base 10 logarithm.

See also:

cpxlog( )	Computes complex natural logarithm

6.14 CPXLONG: Convert Short Complex to Long

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

long cpxlong(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the real part of a short complex value and converts it to long.

Return value:

The long result.

See also: None

6.15 CPXMUL: Short Complex Multiplication

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxmul(a,b)
complex a                   The left-hand value
complex b                   The right-hand value
```

Description:

Multiplies two short complex numbers to form a third.

Return value:

The short complex result of the multiplication.

See also: None

6.16 CPXNEG: Compute the Short Complex Negative

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxneg(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the complex negative of a short complex value.

Return value:

The complex result.

See also: None

6.17 CPXPOL: Short Complex Conversion to Polar

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxpol(a)
complex a                   The value to be converted
```

Description:

Converts a complex number into its polar form. By definition `(r,i)` in Cartesian form is

```
r * exp( i*θ)
```

Return value:

The short complex result of the conversion.

See also: None

6.18 CPXPOW: Raise Short Complex to a Power

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxpow(a,b)
complex a                   The value to be raised to a power
complex b                   The value of the power
```

Description:

Computes the value of a short complex raised to a short complex power.

Return value:

The complex result of the calculation.

See also:

cpxexp( )	Computes a short complex exponential complex
cpxlog( )	Computes a short complex logarithm complex
cpxmul( )	Multiplies two short complex values

6.19 CPXREAL: Compute Real Part of Short Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

float cpxreal(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the real part of a short complex value. The result is double precision.

Return value:

The double precision result.

See also: None

6.20 CPXSIN: Compute the Short Complex Sine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxsin(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the value of the short complex sine of a short complex number. By definition:

$$\sin(a) = \frac{e^{(i * a)} - e^{(-i * a)}}{2i}$$

Return value:

The short complex result.

See also: None

6.21 CPXSROOT: Compute Short Complex Square Root

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxsroot(a)
complex a                   Contains the value to be transformed
```

Description:

Computes the square root of a short complex value.

Return value:

The short complex result.

See also: None

6.22 CPXSUB: Short Complex Subtraction

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

complex cpxsub(a,b)

complex a                   The left-hand value
complex b                   The right-hand value
```

Description:

Subtracts two short complex numbers to form a third.

Return value:

The short complex result of the subtraction.

See also: None

6.23 DPXABS: Compute the Double Complex Absolute Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

double dpxabs(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the value of the double complex absolute value. The result is double precision.

Return value:

The double precision result.

See also: None

6.24 DPXADD: Double Complex Addition

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
```

```
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxadd(a,b)
dcomplex a                  The left-hand value
dcomplex b                  The right-hand value
```

Description:

Adds two double complex numbers to form a third.

Return value:

The double complex result of the addition.

See also: None

6.25 DPXCJG: Compute the Double Complex Conjugate

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxcjg(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the complex conjugate of a double complex value. By definition:

```
      c = conj(a) = a.cr - i * a.ci
```

Return value:

The complex result.

See also: None

6.26 DPXCMP: Double Complex Comparison

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

int dpxcmp(a,b)
```

```
dcomplex a                  The left-hand value
dcomplex b                  The right-hand value
```

Description:

Compares two double complex numbers.

Return value:

A zero if the numbers are the same, else a one.

See also: None

6.27 DPXCOS: Compute the Double Complex Cosine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxcos(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the value of the double complex cosine of a double complex number. By definition:

$$\cos(a) = \frac{e^{(i * a)} + e^{(-i * a)}}{2}$$

Return value:

The double complex result.

See also: None

6.28 DPXCPX: Convert Short Complex to Double Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

typedef struct {
   double dr                The real part of the value
   double di                The imaginary part of the number
```

```
} dcomplex

dcomplex dpxcpx(sng)
complex sng                Contains the value to be transformed
```

Description:

Forms a double complex number whose real and imaginary parts correspond to a single complex number.

Return value:

The double complex result.

See also: None

6.29 DPXDBL: Convert Double Precision to Double Complex

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

typedef struct {
   double cr               The real part of the value
   double ci               The imaginary part of the number
} dcomplex

dcomplex dpxdbl(dbl)
double dbl                 Contains the value to be transformed
```

Description:

Forms a double complex number whose real part is a specified value and whose imaginary part is zero.

Return value:

The double complex result.

See also: None

6.30 DPXDIV: Double Complex Division

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

typedef struct {
   double cr               The real part of the value
   double ci               The imaginary part of the number
} dcomplex

dcomplex dpxdiv(a,b)
dcomplex a                 The numerator
dcomplex b                 The denominator
```

Description:

Divides two double complex numbers to form a third.

Return value:

The double complex result of the division.

See also: None

6.31 DPXDPX: Convert Two Doubles to Double Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
       double cr            The real part of the value
       double ci            The imaginary part of the number
} dcomplex

dcomplex dpxdpx(d1,d2)
double d1                   Contains the real value
double d2                   Contains the imaginary value
```

Description:

Forms a double complex number whose real and imaginary parts are specified values.

Return value:

The double complex result.

See also: None

6.32 DPXEXP: Double Complex Exponential

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
    double cr               The real part of the value
    double ci               The imaginary part of the number
} dcomplex

dcomplex dpxexp(a)
dcomplex a                  The value to be transformed
```

Description:

Computes the double complex exponential of a double complex value.

Return value:

The double complex exponential.

See also: None

6.33 DPXIMA: Compute Imaginary of Double Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

double dpxima(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the imaginary part of a short complex value. The result is double precision.

Return value:

The double precision result.

See also: None

6.34 DPXLOG: Double Complex Natural Logarithm

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxlog(a)
dcomplex a                  The value to be transformed
```

Description:

Computes the double complex natural logarithm of a double complex value.

Return value:

The double complex natural logarithm.

See also: None

6.35 DPXLOG10: Double Complex Base 10 Logarithm

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
       double cr            The real part of the value
       double ci            The imaginary part of the number
} dcomplex

dcomplex dpxlog10(a)
dcomplex a                  The value to be transformed
```

Description:

Computes the double complex base 10 logarithm of a double complex value.

Return value:

The complex base 10 logarithm.

See also:

dpxlog( ) Computes complex natural logarithm

6.36 DPXLONG: Convert Double Complex to Long

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

long dpxlong(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the real part of a double complex value and converts it to long.

Return value:

The long result.

See also: None

6.37 DPXMUL: Double Complex Multiplication

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxmul(a,b)
dcomplex a                  The left-hand value
dcomplex b                  The right-hand value
```

Description:

Multiplies two double complex numbers to form a third.

Return value:

The double complex result of the multiplication.

See also: None

6.38 DPXNEG: Compute the Double Complex Negative

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxneg(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the negative of a double complex value.

Return value:

The double complex result.

See also: None

6.39 DPXPOL: Double Complex Conversion to Polar

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
```

```
} dcomplex

dcomplex dpxpol(a)
dcomplex a                  The value to be converted
```

Description:

Converts a complex number into its polar form. By definition `(r,i)` in Cartesian form is

```
r * exp( i*θ) )
```

Return value:

The double complex result of the conversion.

See also: None

6.40 DPXPOW: Raise Double Complex to a Power

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxpow(a,b)
dcomplex a                  The value to be raised to a power
dcomplex b                  The value of the power
```

Description:

Computes the value of a short complex raised to a short complex power.

Return value:

The complex result of the calculation.

See also:

dpxexp( )	Computes a double complex exponential
dpxlog( )	Computes a double complex logarithm
dpxmul( )	Multiplies two double complex values

6.41 DPXREAL: Compute Real Part of Double Complex

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex
```

```
double dpxreal(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the real part of a short complex value. The result is double precision.

Return value:

The double precision result.

See also: None

6.42 DPXSIN: Compute the Double Complex Sine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxsin(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the value of the double complex sine of a double complex number. By definition:

$$\sin(a) = \frac{e^{(i * a)} - e^{(-i * a)}}{2\ i}$$

Return value:

The double complex result.

See also: None

6.43 DPXSROOT: Compute Double Complex Square Root

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxsroot(a)
dcomplex a                  Contains the value to be transformed
```

Description:

Computes the square root of a double complex value.

Return value:

The double complex result.

See also: None

6.44 DPXSUB: Double Complex Subtraction

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

dcomplex dpxsub(a,b)
dcomplex a                  The left-hand value
dcomplex b                  The right-hand value
```

Description:

Subtracts two double complex numbers to form a third.

Return value:

The double complex result of the subtraction.

See also: None

6.45 FIFAMAX0: FORTRAN Intrinsic Function AMAX0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fifamax0(a1,a2)
long a1                     First value to be compared
long a2                     Second value to be compared
```

Description:

Determines which of two long integer values is the largest.

Return value:

The largest long integer value converted to single precision.

See also: None

6.46 FIFAMIN0: FORTRAN Intrinsic Function AMIN0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fifamin0(a1,a2)
long a1                     First value to be compared
long a2                     Second value to be compared
```

Description:

Determines which of two long integer values is the smallest.

Return value:

The smallest long integer value converted to single precision.

See also: None

6.47 FIFASC50: FORTRAN External Function ASC50

Synopsis:

```
#include "fortran.h"           PROMULA FORTRAN function declarations

void fifasc50(icnt,input,output)
int icnt;                      Number of characters to be converted
void* input;                   Input Radix-50 characters to be converted
void* output;                  ASCII characters return area
```

Description:

The `fifasc50` function converts a specified number of Radix-50 characters to ASCII. If the input word contains an illegal Radix-50 character, a "?" is entered into the ASCII string.

Radix-50 is a special character representation in which up to 3 characters can be encoded and packed into 16 bits. The Radix-50 characters and their corresponding code values are:

Character	Radix-50 Value
space	0
A - Z	1 - 26
a - z	1 - 26
$	27
.	28
0 - 9	30 - 39

The radix-values are stored three characters per word according to the formula:

```
c1 * 40 * 40 + C2 * 40 + c3
```

where the characters are numbered from left to right. If the number of characters is not a multiple of three, it is treated as though it were padded to the right with blanks — i.e., 0.

Note they are called "radix-50" because "40" is "50" when written in octal.

Return value:

None the function is void.

See also: None

6.48 FIFCHAR: FORTRAN Intrinsic Function CHAR

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

char* fifchar(iv)
int iv                      Value to be converted
```

Description:

Converts a numeric display code, or "lexical value" or "collating weight" into its character code. The point of this function is that character values on the host processor are not necessarily the same as those on the machine for which a given FORTRAN program was written. All numeric display code references in a source FORTRAN program are passed through this function either by the translator directly or by the other runtime functions included in this library. Note that if you wish this function to return some value other than the host processor values, you must modify it. Typically this modification would take the form of a lookup table reference.

Return value:

A pointer to a 1-character string containing the character corresponding to the display code.

See also: None

6.49 FIFCOS: FORTRAN Intrinsic Function COS

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fifcos(x)
float x;                    Argument containing the value
```

Description:

Computes the cosine of a floating point argument using single precision.

Return value:

The single precision value as computed above.

See also:

fifsncs( ) Compute sine or cosine

6.50 FIFDATE: External Function DATA

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifdate(cl,ncl)
char* cl;                   Returns the date in string form
int ncl;                    Length of the date return string
```

Description:

Obtains the current date from the operating system.

Return value:

The current date in `mm/dd/yy` form.

See also:

fioconv	General dialect convention flags

6.51 FIFDDIM: FORTRAN Intrinsic Function DDIM

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifddim(a1,a2)
double a1                   First value in difference
double a2                   Second value in difference
```

Description:

Computes a positive difference.

Return value:

If `a1` is greater than `a2` then the value of `a1 - a2` is returned, else zero is returned.

See also: None

6.52 FIFDINT: FORTRAN Intrinsic Function DINT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdint(a)
double a                    Argument containing the value
```

Description:

Truncates its double precision argument to its integer value. In particular:

```
fifdint(a)          = 0 if |a| < 1
```
= the largest integer with the same sign that does not exceed `a` if `|a| >= 1`

Return value:

The double precision value as computed above.

See also: None

6.53 FIFDMAX1: FORTRAN Intrinsic Function DMAX1

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdmax1(a1,a2)
double a1                   First value to be compared
double a2                   Second value to be compared
```

Description:

Determines which of two double precision values is the largest.

Return value:

The largest double precision value.

See also: None

6.54 FIFDMIN1: FORTRAN Intrinsic Function DMIN1

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdmin1(a1,a2)
double a1                   First value to be compared
double a2                   Second value to be compared
```

Description:

Determines which of two double precision values is the smallest.

Return value:

The smallest double precision value.

See also: None

6.55 FIFDMOD: FORTRAN Intrinsic Function

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdmod(num,dem)
double num                  The numerator for the calculation
double dem                  The denominator for the calculation
```

Description:

Computes the value of the remainder of `num` divided by `dem`. If `dem` is zero, the result is zero. For nonzero values the result is calculated as follows:

```
num - (floor  (num/dem  )  * dem)
```

where `floor` is the standard C `floor` function.

Return value:

The value as computed above.

See also: None

6.56 FIFDNINT: FORTRAN Intrinsic Function DNINT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdnint(a)
double a;                   Argument containing the value
```

Description:

Computes the nearest integer to its double precision argument. In particular:

```
fifdnint(a)          = fifdint(a+0.5) if a >= 0.0
          = fifdint(a-0.5) if a <= 0.0
```

where `fifdint` is the FORTRAN intrinsic function `DINT`.

Return value:

The value as computed above.

See also:

fifdint( ) FORTRAN intrinsic function DINT

6.57 FIFDSIGN: FORTRAN Intrinsic Function DSIGN

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fifdsign(mag,sgn)
double mag                  The magnitude for the result
double sgn                  The sign for the result
```

Description:

Returns a value after the transfer of a sign. The result is `|mag|` if `sgn` is at least zero, else it is `-|mag|.`

Return value:

The value as computed above.

See also: None

6.58 FIFEQF: FORTRAN Intrinsic Function EQF

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fifeqf(a,b)
float a;                    First value to be compared
float b;                    Second value to be compared
```

Description:

Compares two floating point numbers to determine if they would be equal in single precision floating point form.

Return value:

The short integer result of the comparison.

See also: None

6.59 FIFEXIT: FORTRAN Exit Subroutine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifexit(numb)
long* numb;                 Message to be displayed at console, i.e: stderr
```

Description:

Exits to the operating system.

Return value:

None, the function exits to the operating system and never returns.

See also: None

6.60 FIFGETAR: FORTRAN Get Command Line Arguments

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifgetarg(k, cl,ncl)
long k;
char* cl;
int ncl;
```

Description:

Copies the `k`th command line argument into a string variable `arg`. The `0`th argument is the command name.

Return value:

None, the function is void.

See also: None

6.61 FIFGETCL: FORTRAN Get Command Line Subroutine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifgetcl(cl,ncl)
char* cl;
int ncl;
```

Description:

Copies the command line parameters as entered by the user into a string variable. The individual parameters are separated by spaces.

Return value:

None, the function is void.

See also: None

6.62 FIFGETENV: FORTRAN Get Value of Environment Variables

Synopsis:

```
#include "fortran.h"                PROMULA FORTRAN function declarations

void fifgetenv(ename,lenin,evalue,lenout)
char* ename                         Name of environment variable
int lenin                           Length of name
char* evalue                        Returns value of environment variable
int lenout                          Length available for variable value
```

Description:

Searches the environment list for a string of the form `ename=value` and returns value in `evalue` if such string is present, otherwise fills `evalue` with blanks.

Return value:

None, the function is void.

See also: None

6.63 FIFHBIT: FORTRAN High Bit Management

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifhbit(chrs,nbyte)
unsigned char* chrs;        Vector of characters
int nbyte;                  Number of bytes in vector
```

Description:

In several dialects of FORTRAN, especially Prime FORTRAN 77, the normal internal character representation is a seven bit code with the 8th bit always set on, while the extended characters have the 8th bit off. The normal external character representations use the opposite conventions. This function converts from one representation to the other by toggling the high order bit in each character of the input vector.

Return value:

None, the function is void.

See also: None

6.64 FIFI2ABS: FORTRAN Intrinsic Function I2ABS

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2abs(a)
short a                     Value whose absolute value is needed
```

Description:

Computes the absolute value of a short integer argument.

Return value:

The short integer result.

See also: None

6.65 FIFI2DAT: FORTRAN External Function I2DATE

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fifi2date(mm,dd,yy)
short* mm;                  Returns month value
short* dd;                  Returns day value
short* yy;                  Returns year value
```

Description:

Obtains the current date from the operating system and stores the month, day, and year in the parameters.

Return value:

None the function is void.

See also: None

6.66 FIFI2DIM: FORTRAN Intrinsic Function I2DIM

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2dim(a1,a2)
short a1                    First value in difference
short a2                    Second value in difference
```

Description:

Computes a positive difference.

Return value:

If `a1` is greater than `a2` then the value of `a1 - a2` is returned, else zero is returned.

See also: None

6.67 FIFI2DINT: FORTRAN Intrinsic Function I2DINT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2dint(a)
double a                    Argument containing the value
```

Description:

Truncates its double precision argument to a short integer value. In particular:

```
fifi2dint(a)  = 0 if |a| < 1
          = the largest integer with the same sign as a that does not exceed a if |a| >= 1
```

Return value:

The value as computed above.

See also: None

6.68 FIFI2MAX0: FORTRAN Intrinsic Function I2MAX0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations
```

```
short fifi2max0(a1,a2)
short a1                    First value to be compared
short a2                    Second value to be compared
```

Description:

Determines which of two short integer values is the largest.

Return value:

The largest short integer value.

See also: None

6.69 FIFI2MIN0: FORTRAN Intrinsic Function I2MIN0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2min0(a1,a2)
short a1                    First value to be compared
short a2                    Second value to be compared
```

Description:

Determines which of two short integer values is the smallest.

Return value:

The smallest short integer value.

See also: None

6.70 FIFI2MOD: FORTRAN Intrinsic Function I2MOD

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2mod(num,dem)
short num                   The numerator for the calculation
short dem                   The denominator for the calculation
```

Description:

Computes the value of the remainder of `num` divided by `dem`. If `dem` is zero, then the result is zero.

Return value:

The value as computed above.

See also: None

6.71 FIFI2NINT: FORTRAN Intrinsic Function I2NINT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2nint(a)
double a                    Argument containing the value
```

Description:

Computes the nearest short integer to its double precision argument. In particular:

```
      fifnint(a)  = fifidint(a+0.5) if a >= 0.0
                  = fifidint(a-0.5) if a <= 0.0
```

where `fifidint` is the FORTRAN intrinsic function `IDINT`.

Return value:

The value as computed above.

See also:

fifidint( )	FORTRAN intrinsic function IDINT

6.72 FIFI2POW: FORTRAN Intrinsic Function I2POW

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifi2pow(a,b)
short a;                    Value to be raised to a power
short b;                    Power to be used
```

Description:

Raises a short integer value to a short integer power.

Return value:

The short integer result.

See also: None

6.73 FIFI2SHF: FORTRAN Intrinsic Function I2SHFT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2shf(a,n)
short a                     Value to be shifted
int n                       Number of positions to be shifted
```

Description:

Shifts the short argument left or right the specified number of bit positions depending upon whether the position count is positive or negative.

Return value:

The short integer result.

See also: None

6.74 FIFI2SIGN: FORTRAN Intrinsic Function I2SIGN

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short fifi2sign(mag,sgn)
short mag                   The magnitude for the result
short sgn                   The sign for the result
```

Description:

Returns a value after the transfer of a sign. The result is `|mag|` if `sgn` is at least zero, else it is `-|mag|`.

Return value:

The value as computed above.

See also: None

6.75 FIFIABS: FORTRAN Intrinsic Function IABS

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifiabs(a)
long a                      Value whose absolute value is needed
```

Description:

Computes the absolute value of a long integer argument.

Return value:

The long integer result.

See also: None

6.76 FIFIARGC: FORTRAN Get Command Line Argument Count

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifiargc( )
```

Description:

Returns the index number of command line arguments — i.e., the number of arguments minus one, where the program name counts as one argument.

Return value:

The number of command line arguments minus one.

See also: None

6.77 FIFIBIT: FORTRAN Intrinsic Function IBIT

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

void fifibit(bits,ibit,ival)
unsigned char* bits;          Bit vector to receive new bit value
int ibit;                     Number of bit to receive value
int ival;                     Bit value to be inserted
```

Description:

Set a specified bit in an arbitrary bit string equal to a specified value. The left-most bit position is numbered 1, with higher bits receiving higher values.

Return value:

None, the function is void.

See also: None

6.78 FIFICHAR: FORTRAN Intrinsic Function ICHAR

Synopsis:

```
#include "fortran.h"        Standard C header file

int fifichar(c1)
char* c1                    Character to be converted
```

Description:

Converts a character code into its numeric display code, or "lexical value" or "collating weight". The point of this function is that character values on the host processor are not necessarily the same as those on the machine for which a given FORTRAN program was written. All character "value" references in a source FORTRAN program are passed through this function either by the translator directly or by the other runtime functions included in this library. Note that if you wish this function to return some value other than the host processor values, then you must modify it. Typically this modification would take the form of a lookup table reference.

Return value:

The character value directly.

See also: None

6.79 FIFIDIM: FORTRAN Intrinsic Function IDIM

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifidim(a1,a2)
long a1                     First value in difference
long a2                     Second value in difference
```

Description:

Computes a positive difference.

Return value:

If `a1` is greater than `a2` then the value of `a1 - a2` is returned, else zero is returned.

See also: None

6.80 FIFIDINT: FORTRAN Intrinsic Function IDINT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifidint(a)
double a                    Argument containing the value
```

Description:

Truncates its double precision argument to a long integer value. In particular:

```
fifidint(a) = 0 if |a| < 1
            = the largest integer with the same sign that does not exceed a if |a| >= 1
```

Return value:

The value as computed above.

See also: None

6.81 FIFINDEX: FORTRAN Intrinsic Function INDEX

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations
```

```
int fifindex(s,ns,c,nc)
char* s                    Main string
int ns                     Length of main string
char* c                    Substring
int nc                     Length of substring
```

Description:

Returns the location of a substring within a string. Note that the arguments are two FORTRAN style strings, which means that the length of each string must also be sent.

Return value:

If the substring occurs within the main string, the result is an integer indicating the starting position (relative to 1) of the first occurrence of the substring within the main string. If the substring does not occur within the main string, a zero is returned.

See also: None

6.82 FIFIPOW: FORTRAN Intrinsic Function IPOW

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

long fifipow(a,b)
long a;                    Value to be raised to a power
long b;                    Power to be used
```

Description:

Raises a long integer value to a long integer power.

Return value:

The long integer result.

See also: None

6.83 FIFISHF: FORTRAN Intrinsic Function ISHFT

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

long fifishf(a,n)
long a                     Value to be shifted
int n                      Number of positions to be shifted
```

Description:

Shifts the long argument left or right the specified number of bit positions depending upon whether the position count is positive or negative.

Return value:

The long integer result.

See also: None

6.84 FIFISIGN: FORTRAN Intrinsic Function ISIGN

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifisign(mag,sgn)
long mag                    The magnitude for the result
long sgn                    The sign for the result
```

Description:

Returns a value after the transfer of a sign. The result is `|mag|` if `sgn` is at least zero, else it is `-|mag|.`

Return value:

The value as computed above.

See also: None

6.85 FIFMAX0: FORTRAN Intrinsic Function MAX0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifmax0(a1,a2)
long a1                     First value to be compared
long a2                     Second value to be compared
```

Description:

Determines which of two long values is the largest.

Return value:

The largest long value.

See also: None

6.86 FIFMAX1: FORTRAN Intrinsic Function MAX1

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifmax1(a1,a2)
float a1                    First value to be compared
float a2                    Second value to be compared
```

Description:

Determines which of two double precision values is the largest.

Return value:

The largest value converted to a long integer.

See also: None

6.87 FIFMIN0: FORTRAN Intrinsic Function MIN0

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifmin0(a1,a2)
long a1                     First value to be compared
long a2                     Second value to be compared
```

Description:

Determines which of two long values is the smallest.

Return value:

The smallest long value.

See also: None

6.88 FIFMIN1: FORTRAN Intrinsic Function MIN1

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifmin1(a1,a2)
float a1                    First value to be compared
float a2                    Second value to be compared
```

Description:

Determines which of two double precision values is the smallest.

Return value:

The smallest value converted to a long integer.

See also: None

6.89 FIFMOD: FORTRAN Intrinsic Function MOD

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

long fifmod(num,dem)
```

`long num`	The numerator for the calculation
`long dem`	The denominator for the calculation

Description:

Computes the value of the remainder of `num` divided by `dem`. If `dem` is zero, then the result is zero.

Return value:

The value as computed above.

See also: None

6.90 FIFNEF: FORTRAN Intrinsic Function NEF

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`int fifnef(a,b)`	
`float a;`	First value to be compared
`float b;`	Second value to be compared

Description:

Compares two floating point numbers to determine if they would be unequal in single precision floating point form.

Return value:

The short integer result of the comparison.

See also: None

6.91 FIFNINT: FORTRAN Intrinsic Function NINT

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`long fifnint(a)`	
`double a`	Argument containing the value

Description:

Computes the nearest integer to its double precision argument. In particular:

```
fifnint(a)  =  fifidint (a + 0.5) if a ≥ 0.0
            =  fifidint (a - 0.5) if a ≤ 0.0
```

where `fifidint` is the FORTRAN intrinsic function `IDINT`.

Return value:

The value as computed above.

See also:

fifidint( )	FORTRAN intrinsic function IDINT

6.92 FIFRAD50: FORTRAN External Function IRAD50

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

int fifrad50(icnt,input,output)
int icnt;                     Number of characters to be converted
void* input;                  Input characters to be converted
void* output;                 Radix-50 characters return area
```

Description:

The `fifrad50` function converts a specified number of ASCII characters to Radix-50. Conversion stops on the first non-Radix-50 character encountered in the input, or when the specified number of characters have been converted.

Radix-50 is a special character representation in which up to three characters can be encoded and packed into 16 bits. The Radix-50 characters and their corresponding code values are:

Character	Radix-50 Value
space	0
A - Z	1 - 26
a - z	1 - 26
$	27
.	28
0 - 9	30 - 39

The radix-values are stored three characters per word according to the formula:

c1 * 40 * 40 + C2 * 40 + c3

where the characters are numbered from left to right. If the number of characters is not a multiple of three, then it is treated as though it were padded to the right with blanks — i.e., 0.

Note they are called "radix-50" because "40" is "50" when written in octal.

Return value:

The number of characters actually converted.

See also: None

6.93 FIFRBIT: FORTRAN Intrinsic Function RBIT

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

void fifrbit(bits,nbyte)
unsigned char* bits;          Bit vector to be reversed
int nbyte;                    Number of bytes in bit vector
```

Description:

Reverses the order of the bytes in a bit string.

Return value:

None, the function is void,

See also: None

6.94 FIFSIN: FORTRAN Intrinsic Function SIN

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fifsin(x)
float x;                    Argument containing the value
```

Description:

Computes the sine of a floating point argument using single precision.

Return value:

The value as computed above.

See also:

fifsncs( )	Compute sine or cosine

6.95 FIFSNCS: FORTRAN Single Precision Sine/Cosine

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fifsncs(x,cosflag)
float x;                    Argument containing the value
int cosflag;                Perform cosine flag
```

Description:

Computes the sine or cosine of a single precision value which has been promoted to double via the C promotion rules.

Return value:

The double precision value as computed above.

See also: None

6.96 FIFSTRGV: FORTRAN String Value Conversion

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

char* fifstrgv(str,nc)
char* str;                  String to be converted
int nc;                     Number of characters in string
```

Description:

Copies a character string into a local buffer and then pads that buffer to 8 characters with blanks, so that the string can be compared to a character string which has been hidden in a noncharacter variable. Note that such hiding always pads the numeric with blanks out to its natural size. This function is needed for situations like

```
F(K .EQ. 'Y')
```

where `K` is a noncharacter variable. The translation of this would be

```
if ( k == *(typeof(k)*) fifstrv ("Y",1))
```

Return value:

A pointer to a buffer containing the string value.

See also: None

6.97 FIFSYSTM: FORTRAN External Function SYSTEM

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fifsystm(cl,ncl)
char* cl                    String containing command to be executed
int ncl                     Length of the execution string
```

Description:

Executes the command contained in the character string and then resumes execution of the current program. The problem addressed by this function is that "system" requires a nul-terminated string; while the string received is blank-padded with no terminator.

Return value:

A system-dependent integer status from the command. In UNIX systems, the status return is the value returned by `exit`.

See also: None

6.98 FIFTAN: FORTRAN Intrinsic Function TAN

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

float fiftan(x)
float x;                    Argument containing the value
```

Description:

Computes the tangent of a floating point argument.

Return value:

The value as computed above.

See also: None

6.99 FIFTIME: FORTRAN External Function TIME

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiftime(cl,ncl)
char* cl;                   Returns the time in string form
int ncl;                    Length of the time return string
```

Description:

Obtains the current time from the operating system.

Return value:

The current time in `hr:mi:sec` form.

See also: None

6.100 FIFXBIT: FORTRAN Intrinsic Function XBIT

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fifxbit(bits,ibit)
unsigned char* bits;        Bit vector to receive new bit value
int ibit;                   Number of bit to receive value
```

Description:

Extract a specified bit value from an arbitrary bit string. The left-most bit position is numbered 1, with higher bits receiving higher values.

Return value:

The function returns a one or a zero depending upon whether the bit specified is on or off.

See also: None

6.101 FIFXCREP: FORTRAN Extended Character Representation

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations
```

```
char* fifxcrep(chrs)
char* chrs;                Vector of characters
```

Description:

In several dialects of FORTRAN, especially Prime FORTRAN 77, the normal internal character representation is a seven bit code with the 8th bit always set on, while the extended characters have the 8th bit off. The normal external character representations use the opposite conventions. This function converts string constants to the extended representation to make them compatible with other "internal" characters.

Return value:

A pointer to the result string.

See also: None

6.102 FIOBACK: Backspace a FORTRAN File

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

int fioback( )
```

Description:

This function backspaces the file associated with the "current" FORTRAN file as specified in the global variable `fiocurf`. For files with fixed record lengths, this operation involves moving backwards in the file by the length of one record. For text files, this means moving backwards in the file to the position immediately following the second carriage-return line-feed character pair which immediately precedes the current record. For other types of files, no backspace is possible. Note that if the file is currently positioned at its beginning, no operation is performed.

Return value:

A zero if the backspace was successful, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( ) Performs standard error processing

6.103 FIOBFOUT: Business Format Output

Synopsis:

```
#include "fortran.h"       PROMULA FORTRAN function declarations

void fiobfout(value,bfmt)
double value;              Value to be converted
char* bfmt;                Business formatting descriptor string
```

Description:

This utility function controls the conversion of a double precision value to string form under the control of a business formatting descriptor string. The length of the string determines the field width of the final display. If this width is too small for the number, then the output will be a string of asterisks filling the field. Valid characters for the string are:

+ * – Z $ # , . CR

The use of these valid characters is explained below:

Plus (+):

If only the first character is +, then the sign of the number (+ or -) is entered in the leftmost portion of the field (fixed sign). If the string begins with more than one + sign, they will be replaced by blanks and the sign of the number (+ or -) will be printed in the field position immediately to the left of the first printing character of the number (floating sign). If the rightmost character of the string is +, then the sign of the number (+ or -) will be printed in that field position following the number (trailing sign).

Minus (-):

The minus sign behaves the same as a plus sign except that a space (blank) is entered instead of a + if the number is positive (plus sign suppression).

Dollar sign ($):

A dollar sign may at most be preceded in the string by an optional fixed sign. A single dollar sign will cause a $ to be printed in the corresponding position in the output field (fixed dollar). Multiple dollar signs will be replaced by printing characters in the number, and a single $ will be printed in the position immediately to the left of the leftmost printing character of the number (floating dollar).

Asterisk (*):

Asterisks may be preceded only by an optional fixed sign and/or a fixed dollar. Asterisks in positions used by digits of the number will be replaced by those digits. The remainder will be printed as asterisks (field filling).

Zed (Z):

If the digit corresponding to a Z in the output number is a leading zero, a space (blank) will be printed in that position; otherwise, the digit in the number will be printed (leading-zero suppression).

Number sign (#):

The number sign indicates a digit position not subject to leading-zero suppression: the digit in the number will be printed in its corresponding portion whether zero or not (zero non-suppression).

Decimal point (.):

A decimal point indicates the position of the decimal point in the output number. Only the # sign and either trailing signs or credit (CR) may follow the decimal point.

Comma (,):

Commas may be placed after any leading character, but before the decimal points. If a significant character of the number (not a sign or dollar) precedes the comma, a comma will be printed in that position. If not preceded by a significant character, a space will be printed in this position unless the comma is in an asterisk field. In that case an * will be printed in that position.

Credit (CR):

The characters CR may only be used as the last two (rightmost) characters of the string. If the number is positive, two spaces will be printed following it. If negative, the letters CR will be printed.

Return value:

None, the function is void. The global variable `fiocrec` is updated to contain the new field, and the variable `fionchar` is updated to reflect the new character count in the coded communications record.

See also: None

6.104 FIOCLOSE: Close Current FORTRAN File

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fioclose( )
```

Description:

This function closes the file associated with the "current" FORTRAN file as specified in the global variable `fiocurf`.

Return value:

A zero if the close was successful, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( ) Performs standard error processing

6.105 FIOCPATH: Convert Pathname

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiocpath(pathname,cinfo)
char* pathname              Pathname to be converted
unsigned char* cinfo        Conversion information table
```

Description:

Converts a pathname as it appears in an OPEN or open-related statement into a form compatible with the target platform. The actual conversion information is stored in a conversion information table which has the following structure:

Byte	Description of content
0	Directory separation character
1	Exclude directories from pathname flag
2	Case conversion code (0 = none, 1 = toupper, 2 = tolower)
3+	Other conversions string
4+	Prefix to be added to name (length,characters)
5+	Conversions list (two Pascal-style strings)

6 Nul termination byte

The conversion information table itself is constructed via function "fiorpath".

Return value:

None, the function is void; however, the content of the `pathname` parameter is altered to reflect the conversion.

See also:

fiorpath( )	Reads the path conversion information used here
fioshl( )	Shift string left
fioshr( )	Shift string right

6.106 FIODTOS: Convert Double Value to String

Synopsis:

```
#include "fortran.h"                    PROMULA FORTRAN function declarations

void fiodtos(val,ndig,pdecpt,psign,dspdig)
double val                              Value to be converted
int ndig                                Number of digits to produce
int* pdecpt                             Returns position of decimal point
int* psign                              Returns the sign of the value
char* dspdig                            Returns the string produced
```

Description:

ANSI C expects that all conversions of floating point values to string be performed via the `sprintf` function. Though this can be done, most generalized applications prefer to perform their own editing operations, and require only a raw conversion be performed. This function performs that conversion using whatever facilities are available with a particular platform.

This function converts the floating point number `val` to a character string. It stores precisely `ndig` digits in `dspdig` followed by a null-byte. If the number of digits in `val` exceeds `ndig`, the last digit is rounded; if the number of digits is less than `ndig`, then `dspdig` is padded with zeros. The parameter `pdecpt` points to an integer value giving the position of the decimal point with respect to the beginning of the string; and `psign` returns zero if `val` is positive; else one.

Return value:

None, the function is void.

See also: None

6.107 FIOERROR: Perform FORTRAN I/O Error Processing

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fioerror(clear)
int clear                   Should error control be cleared?
```

Description:

If the FORTRAN I/O runtime system encounters an error, it sets an error code and calls this function. This function either sets an error return value or exits to the operating system with an error message. In the case where an error code is returned to the calling function, the parameter `clear` specifies whether or not the error processing control variables should be cleared prior to the return.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiocurf	Pointer to current FORTRAN file
fioitos( )	Converts an integer to a string
fioerch	Specifies presence of error checking
fioier	Code for actual error encountered
fiostat	Returns an error code or zero

6.108 FIOFDATA: FORTRAN File Data

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

void fiofdata(option,str,ns)
int option                    Specifies which data is being specified
char* str                     String information
int ns                        String length or integer information
```

Description:

This function is used to specify the various file data options associated with the current FORTRAN file structure. The particular data being specified is defined by the `option` parameter as follows:

Option	Description of data
1	Specifies the file name to be assigned to the file. This name may have no more than 39 characters. If the name is NULL or if it is all blank, then no name is assigned. If the file is opened with no name assigned, then the open function will request a name from `stdinp` via `stdout`.
2	Specifies the status of the file. Only the first character of the `string` parameter is used as follows:

Char	Meaning
O,o	Old — the file already exists, do not create it.
N,n	New — the file does not exist, create it even if this means destroying an existing file with the same name.
S,s	Scratch — the same as new, except that the file is removed when it is closed.
U,u	Unknown — if the file exists it is opened, if it does not exist then it is created.

3 Specifies the access type of the file. Only the first character of the `string` parameter is used as follows:

Char	Meaning
S,s	Sequential — the file is opened for sequential access.
D,d	Direct — the file is opened for direct access.

4 Specifies the form of the file. Only the first character of the `string` parameter is used as follows:

Char	Meaning
F,f	the records are formatted
U,u	the records are unformatted

5 Specifies the record size for the file as contained in the parameter `ns`. The parameter `string` is ignored.

6 Specifies the convention for treating blanks in numeric input fields. Only the first character of the `string` parameter is used as follows:

Char	Meaning
N	NULL — blank characters in numeric formatted input fields are ignored, except that a field of all blanks is zero.
Z	ZERO — Blanks are treated as zeros.

7 Specifies that the file is readonly.

8 Specifies that the file is to be open for shared access.

9 Specifies the record type.

10 Specifies the carriage control conventions.

11 Specifies a pointer to the associated variable.

12 Specifies the maximum number of records on the file.

13 Specifies a pointer to receive the number of the unit assigned to the file.

14 Specifies the associated buffer address along with its size.

15 Specifies a buffer block size.

Return value:

None, this function is void.

See also: None

6.109 FIOFEND: End Format Processing

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations
void fiofend( )
```

Description:

Clears the format control variables to end processing with the current format.

Return value:

None, the function is void.

See also: None

6.110 FIOFFLD: Get Next Free-Form Field

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fioffld( )
```

Description:

This function locates the start of the next unit of information when free-form reads are being performed. It skips over blanks and commas. If an end-of-record is encountered, then it reads the next physical record from the current file. The major complexity in this routine has to do with repeat counts and null-values. Note that when a "/" character is encountered, all remaining values are set to null.

Return value:

The function returns a 1 if there is a non-null value to be read. In this case, `fioicol` points to the first significant character of this value. A zero is returned if a null value is to be read. From the standpoint of the calling function, this means that the corresponding input value is to remain unchanged.

See also:

fiospace( )	Skip over white space

6.111 FIOFINI: Initialize A FORTRAN Format

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiofspec = 0            Current format specification
char* fiosadr = NULL        Format string location
char* fiocfmt = NULL        Current format position
char** fiofrmt = NULL       Current format address list
int fiofstat = 0            Current processing state
int fioifmt = 0             Number of current format entry
int fioiresc = 0            Number of current rescan entry
int fiomaxc = 0             Maximum characters in output line
int fionfmt = 0             Number of entries in format list
int fionpren = 0            Parenthetical nesting
char* fiorscan = NULL       Current rescan position
int fiorscnt = 0            Rescan count value
char fiobfmt[40]            Business formatting string
char* fiovfmt = NULL        Start of variable format
int fiovflen = 0            Length of variable format

int fiofini(fmt,nfmt)
char** fmt                  Pointer to the format or format pointers
int nfmt                    Number of format pointers or type
```

Description:

This function initializes the FORMAT environment needed by the FORTRAN style input/output statements. There is a minor problem associated with the processing of FORTRAN style FORMATS in C, having to do with "maximum string length". In theory, since a C string consists of a sequence of nonzero characters terminated by a zero byte, it can be of any

length desired. Unfortunately, all C compilers place a limit on the maximum length that a string constant may have, typically 256. It is not at all unusual for FORMAT specification strings to be very long, and they are typically defined as constants at compile time. Fortunately, though length of an individual string constant is limited, C has a very neat notation for defining a constant set of pointers to constant strings. Thus, a long format string can be written very conveniently as a sequence of individual lines as follows:

```
static char fmt01[] = {
       "(T2,20A4,//,T2,'MASS (M) UNITS = ',2A4,/,"
       "T2,'LENGTH (L) UNITS = ',2A4,/,"
       "T2,'TIME (T) UNITS = ',2A4,//)"
};
```

Using this notation, indefinitely long FORMAT specification strings can be written as a sequence of blocked lines. Note that the FORMAT routines ignore the breaks between the lines, so they may be broken in any way. Simple FORMAT specification strings may, of course, still be written as standard string constants such as the following:

```
static char fmt02[] = "(F10.0)";
```

The FORMAT specification string itself has the same content as a FORMAT would have in a standard FORTRAN environment. It must begin with a left-parenthesis and end with a right-parenthesis. See the general discussion of FORTRAN I/O capabilities for a detailed description of the format specifications. There are two parameters to be supplied for this function as follows:

Name	Description of use
`fmt`	Points to the format control string. As discussed above, it may either be a pointer to a series of strings or it may be a simple string pointer. It may also be a NULL, meaning either that free-format is to be used or that a "binary" operation is to be performed.
`nfmt`	Provides additional information about the above. If `fmt` is NULL and `nfmt` is zero, then an unformatted binary type operation is being performed. If `fmt` is NULL and `nfmt` is nonzero, then a free-form "formatted" operation is being performed. If `fmt` is not NULL and `nfmt` is zero, then a single string format is being processed; else the specification consists of a sequence of `nfmt` lines.

Once this function is called, the format specification string remains in effect until either this function is called again, or function `fiofend` is called.

Return value:

A zero if the format appears well-formed, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Performs standard error processing
fiofwsp( )	Skips over white space in the format

6.112 FIOFINP: Formatted Input

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`void fiofinp(context)`	
`int context`	Context of call 0 = value, 1 = end

Description:

Performs nonvariable related formatted input functions until an end-of-format or a variable related specification is encountered. The actual operations performed by this function are as follows:

Specification	Code	Description
nH	1	Display Hollerith string
"c1..cn"	1	Display delimited string
nX	2	Skip right n places
TRn	2	Skip right n places
Tn	3	Move to position n
TLn	4	Skip left n places
SS	5	Set the plus sign to a space
SP	6	Set the plus sign to a +
BN	7	Set blanks to null
BZ	8	Set blanks to zero
/	9	Physically write the current line
nP	10	Set the floating scale factor to n

Return value:

Note the function is void; however, the global variable `fioier` may be set to an error code if a problem is encountered.

See also:

fionxtf( )	Get next format specification
fiorchk( )	Check fixed input field
fiortxt( )	Read next text record

6.113 FIOFINQU: Inquire About File Data

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

void fiofinqu(option,str,ns)
int option                    Specifies data being inquired about
char* str                     Information to be returned
int ns                        String length or integer information
```

Description:

This function is used to inquire about the various file data options associated with the current FORTRAN file structure. The particular data being inquired about is defined by the `option` parameter as follows:

Option	Description of data
1	Inquire about status of file
2	Inquire about existence of file
3	Inquire as to connection status of file
4	Inquire as to file's external unit number
5	Inquire whether file has a name
6	Inquire for name of file
7	Inquire as to files access method

8	Inquire if file can be accessed sequentially
9	Inquire if file can be directly accessed
10	Inquire if file is formatted
11	Inquire if file can be opened as a text file
12	Inquire if file can be opened as binary
13	Inquire about file record length
14	Inquire about file current record number
15	Inquire about file current blank convention

Return value:

None, this function is void.

See also: None

6.114 FIOFMTV: Compute FORMAT Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiofmtv( )
```

Description:

Computes the value of an integer constant in the FORMAT statement, and updates the current format position so that it points to the first nonblank character beyond the end of the value. Note that if the current character in the format is nonnumeric when this function is called, then this function does not move the current position and returns a zero value.

Return value:

The value of the integer constant or a zero if there was no integer constant.

See also:

fiofwsp( )	Skip white space in format

6.115 FIOFOUT: Formatted Output Operations

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiofout(context)
int contex                  Context of call 0 = value, 1 = end
```

Description:

Performs nonvariable related formatted output functions until an end-of-format or a rescan or variable related specification is encountered. The actual operations performed by this function are as follows:

Specification	Code	Description
nH	1	Write characters from a Hollerith string
"c1..cn"	1	Write characters from a delimited string

nX	2	Skip right n places
TRn	2	Skip right n places
Tn	3	Move to position n
TLn	4	Skip left n places
SS	5	Set the plus sign to a space
SP	6	Set the plus sign to a +
BN	7	Set blanks to null (no operation for output)
BZ	8	Set blanks to zero (no operation for output)
/	9	Physically write the current record
nP	10	Set the floating scale factor to n
)	11	End-of-Format write current record

Return value:

None, the function is void; however, the global variable `fioier` may be set to an error code if a problem is encountered.

See also:

fionxtf( )	Get next format specification
fiowtxt( )	Write text record

6.116 FIOFVINQ: Inquire About File Value

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
long fiofvinq(option)
int option
```
Specifies data being inquired about

Description:

This function is used to inquire about the various file data options associated with the current FORTRAN file structure which return an integer or logical value. The particular value being inquired about is defined by the `option` parameter as follows:

Option	Description of data
2	Inquire about existence of file
3	Inquire as to connection status of file
4	Inquire as to file's external unit number
5	Inquire whether file has a name
13	Inquire about file record length
14	Inquire about file current record number

Note that this function is needed as distinct from the generic file information function because type conversions are required on the returned value.

Return value:

The requested logical or integer value.

See also:

fiofinqu( )	Generic file information function

6.117 FIOFWSP: Skip Format White Space

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiofwsp( )
```

Description:

This function is a utility used by the FORMAT processing functions to skip over white space within the specification string. White space consists of blanks and boundaries between the independent lines of the specification.

Return value:

None, the function is void. The effect of its processing is reflected in the various global format control variables.

See also: None

6.118 FIOINTU: Establish FORTRAN Internal Unit

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiointu(intu,rsize,action)
char* intu                  Pointer to internal storage
int rsize                   Record size
int action                  Specifies read = 1 or write = 0
```

Description:

When some action is to be performed on an internal unit, typically a character storage area, this function initializes a file structure to point to this internal unit, and makes this file structure the current one. Note that if the record size is zero, the internal unit is a set of pointers to C-style strings and not FORTRAN style strings.

The form of the creation of the INTERNAL depends upon the type of the action to be performed. This action code is as follows:

Code	Action to be performed
0	An internal file is to be opened with the specified storage area as its starting address. The size parameter specifies the overall size of the area. If there is an open structure currently using this starting address then it is simply rewound.
1	An internal file with this starting address has been created. If it cannot be found, then an error has occurred.
2	A coded read is to be performed. If there is no structure defined, then create one which will be removed at the end of the operation (this is the standard behavior).
3	A binary read is to be performed. If there is no structure defined, then create one which will be removed at the end of the operation.

4 A coded write is to be performed. If there is no structure defined, then create one which will be removed at the end of the operation (this is the standard behavior).

5 A binary write is to be performed. If there is no structure defined, then create one which will be removed at the end of the operation.

6 A miscellaneous operation is to be performed. If there is no structure defined, then an error has occurred.

7 A simple inquiry is being made. If there is no structure defined, then create one which will be removed at the end of the operation.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Performs standard error processing
fiortxt( )	Read text record

6.119 FIOITOS: Convert Integer to String

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

char* fioitos(numb)
int numb                    Value to be converted
```

Description:

Converts an integer number to a character string.

Return value:

A pointer to the string result.

See also: None

6.120 FIOLREC: Position a FORTRAN File on a Record

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiolrec(irec)
long irec;                  Record number desired
```

Description:

This function positions the file associated with the "current" FORTRAN file as specified in the global variable `fiocurf` at the beginning of a specified record.

Return value:

A zero if the positioning was successful, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.121 FIOLTOS: Convert Long Integer to String

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fioltos(value)
long value                  Value to be converted
```

Description:

Converts a long integer value to display form and stores it at the current position in the coded communications record right-justified in a fixed length field. In particular, the output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise. If the number of significant digits and sign required to represent the value is less than the specified width, the unused leftmost portion of the field is filled with blanks. If it is greater than the width, asterisks are entered instead of numeric digits. If a minimum digit count is specified, the output field consists of at least that many digits, and is zero-filled as necessary. If the minimum digit count is zero, and the value is zero, then the field is simply blank filled, regardless of any sign control in effect.

The parameter `value` contains the value to be converted. The conversion control parameters are specified via global variables.

Return value:

None, the function is void. The global variable `fiocrec` is updated to contain the new field, and the variable `fionchar` is updated to reflect the new character count of the coded communication record.

See also:

fioitos( )	Convert integer to string
fioshr( )	Shift display string right

6.122 FIOLUN: Establish FORTRAN Unit Number

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiolun(lun,action)
int lun                     Logical unit number of file
int action                  Action code for subsequent use
```

Description:

Before any action can be performed on a FORTRAN file, the logical unit number must be associated with an existing FORTRAN file structure. If there is no already existing structure for the unit number, then this function will attempt to create one. The form of this creation depends upon the type of the action to be performed. This action code is as follows:

Code	Action to be performed
0	A file is to be opened with this logical unit number, if there is an open structure currently associated with this number, close the file.
1	A structure for this logical unit number has been created. If it cannot be found, then an error has occurred.
2	A coded read is to be performed. If there is no structure defined, then create one and open the file.
3	A binary read is to be performed. If there is no structure defined, then create one and open the file.
4	A coded write is to be performed. If there is no structure defined, then create one and create the file.
5	A binary write is to be performed. If there is no structure defined, then create one and create the file.
6	A miscellaneous operation is to be performed. If there is no structure defined, then create one and open the file.
7	A simple inquiry is being made.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioclose( )	Closes the current FORTRAN file
fioerror( )	Performs standard error processing
fioopen( )	Opens the current FORTRAN file
fiortxt( )	Read next text record

6.123 FIONAME: Establish FORTRAN Unit by Name

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fioname(strg,ns)
char* strg                  Name of file
int ns                      Number of characters in file name
```

Description:

The FORTRAN INQUIRE statement allows the user to inquire about file status either via its logical unit number or via its name. If the name reference is being used, then this function is called. If there is no already existing structure for a unit with this name, then this function will attempt to create one.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fifindex( )	Find one substring in another
fioerror( )	Performs standard error processing

6.124 FIONXTF: Get Next Format Specification

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fionxtf( )
```

Description:

Gets the next format specification from the format list and sets the external variables (`fiofspec, fioiwd`, and `fiondec`) to indicate what it is. The above variables are set as follows:

Specification	Type	fioiwd	fiondec
free form	0	0	0
nH	1	n	0
"c1..cn"	1	n	"
'c1..cn'	1	n	'
c1..cn	1	n	*
nX	2	n	--
TRn	2	n	--
Tn	3	n	--
TLn	4	n	--
SS	5	--	--
SP	6	--	--
BN	7	--	--
BZ	8	--	--
/	9	--	--
nP	10	n	--
)	11	---	--
:	-1	---	--
$	-2	---	--
Aw	12	w	--
Lw	13	w	--
Iw	14	w	--
Fw.d	15	w	d
Dw.d	16	w	d
Ew.d	17	w	d
Gw.d	18	w	d
B'ssss'	19	--	--

Return value:

None, the function is void.

See also:

fioerror( )	Do requested error processing
fiofmtv( )	Get constant format value
fiofwsp( )	Skip over white space

6.125 FIOOPEN: Open Current FORTRAN File

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fioopen( )
```

Description:

This function opens the file associated with the "current" FORTRAN file as specified in the global variable `fiocurf`.

Return value:

A zero if the open went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioitos( )	Converts a short integer to a string
fioerror( )	Performs standard error processing

6.126 FIORALPH: Read Alphabetic Information

Synopsis:

```
#include "fortran.h"                PROMULA FORTRAN function declarations

void fioralph(alpha,nalpha,nfield)
char* alpha                         Character string
int nalpha                          Length of string
int nfield                          Width of field
```

Description:

This function blank fills a character string and then reads characters from the current input record into the string. Read characters are left-justified in the string. Any characters beyond the end of the string are discarded. If input is free-form and if the characters read begin with a single quote, then the material within the quoted list is entered into the character string, with " reducing to a single quote.

Return value:

None, the function is void. The effect of its processing is reflected in the various global format control variables.

See also:

fiortxt( )	Read next text record
ftnxcons(.)	Process Exact representation constant
fiorchk( )	Check fixed-form input field

6.127 FIORBIV: FORTRAN Read Binary Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiorbiv(value,nvalue)
void* value                 Points to values being read
int nvalue                  Number of bytes to be read
```

Description:

This function reads binary values from a file.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.128 FIORCHK: Check Fixed-Form Input Field

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiorchk(nfield)
int nfield
```

Description:

This function controls the physical reading of text records. If the logical end-of-record has been reached, the next physical record is read. If a physical end-of-record has been reached prior to the logical end-of-record, the physical end-of-record is extended by padding the record with blanks.

Return value:

None, the function is void. The effect of its processing is reflected in the various global format control variables.

See also: None

6.129 FIORDB: Read FORTRAN Boolean Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordb(bool,nval)
unsigned short* bool        The values to be read
int nval                    The number of values to be read
```

Description:

Reads a vector of Boolean (short logical) values from the current input file in accordance with the current format specification. Note that in this function each individual Boolean value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fioffld( )	Get next free-form field

fiofinp( )	Process input format specifications
fioralph( )	Read alphabetic information
fiorchk( )	Check fixed read field

6.130 FIORDC: Read FORTRAN Character Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiordc(c,nval)
char* c
int nval
```
Points to characters to be read
Number of characters to read

Description:

Reads a vector of character values from the current input file in accordance with the current format specification. In this implementation signed `char` is derived from the nonstandard FORTRAN types BYTE or INTEGER*1. Therefore, `I` formatting conventions are assumed. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do specified error processing
fiordi( )	Read short integer value

6.131 FIORDD: Read FORTRAN Double Precision Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiordd(value,nval)
double* value
int nval
```
Points to values to be read
Number of values to be read

Description:

Reads a vector of double precision floating point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fiostod( )	Convert string to double
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information
fiorchk( )	Check current fixed input field

6.132 FIORDF: Read FORTRAN Floating Point Values

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiordf(value,nval)
float* value
int nval
```
Points to values to be read
Number of values to be read

Description:

Reads a vector of single precision floating point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fiostod( )	Convert string to double
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information
fiorchk( )	Check current fixed input field

6.133 FIORDI: Read FORTRAN Short Integer Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiordi(value,nval)
short* value
int nval
```
Vector of values
Number of values to be read

Description:

Reads a vector of short fixed point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fiostod( )	Convert string to double
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information
fiorchk( )	Check current fixed input field

6.134 FIORDL: Read FORTRAN Long Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordl(value,nval)
long* value                 Vector of values
int nval                    Number of values to be read
```

Description:

Reads a vector of long fixed point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fiostod( )	Convert string to double
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information
fiorchk( )	Check current fixed input field

6.135 FIORDS: Read FORTRAN String

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

int fiords(str,nstring,nval)
char* str                       Points to start of strings
int nstring                     Length of each string
int nval                        Number of strings to be read
```

Description:

Reads a sequence of fixed length strings, stored one after another, from the current input file in accordance with the current format specification. Each string is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information

6.136 FIORDT: Read FORTRAN Truth Value Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordt(bool,nval)
unsigned long* bool         The values to be read
int nval                    The number of values to be read
```

Description:

Reads a vector of truth values (long logical values) from the current input file in accordance with the current format specification. Note that in this function each individual truth value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fiofinp( )	Get next formatted input specification
fioffld( )	Get next free-form input field
fioralph( )	Read alphabetic information
fiorchk( )	Check current fixed input field

6.137 FIORDU: Read FORTRAN Unsigned Char Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordu(c,nval)
unsigned char* c            Points to characters to be read
int nval                    Number of characters to read
```

Description:

Reads a vector of character values from the current input file in accordance with the current format specification. In this implementation `unsigned char` is derived from the nonstandard FORTRAN type LOGICAL*1. Therefore, L formatting conventions are assumed. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

int fiordb( )	Read short Boolean value
int fioerror( )	Perform FORTRAN I/O error processing

6.138 FIORDX: Read FORTRAN Complex Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordx(value,nval)
complex* value;             Points to values to be read
int nval;                   Number of values to be read
```

Description:

Reads a vector of single precision complex values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fioffld( )	Get next freeform input field
fiordf( )	Read floating point values
fiospace( )	Skip white-space in record
fiostod( )	Convert string to double

6.139 FIORDZ: Read FORTRAN Double Complex Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiordz(value,nval)
```

`dcomplex* value;`	Points to values to be read
`int nval;`	Number of values to be read

Description:

Reads a vector of double precision complex values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O error processing
fioffld( )	Get next freeform input field
fiordd( )	Read floating point values
fiospace( )	Skip white-space in record
fiostod( )	Convert string to double

6.140 FIOREC: Position a FORTRAN File on a Record

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations

```
int fiorec(irec)
```

`int irec`	Record number desired

Description:

This function merely records a record number at which the FORTRAN file about to be accessed is to be positioned.

Return value:

None, the function is void.

See also: None

6.141 FIOREW: Rewind a FORTRAN File

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations

```
int fiorew( )
```

Description:

This function rewinds the file associated with the "current" FORTRAN file as specified in the global variable `fiocurf`.

Return value:

A zero if the rewind was successful, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.142 FIORLN: Read FORTRAN End-of-Line

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiorln()
```

Description:

Completes the current read operation by flushing the current format statement and by setting the current record controls to the end of the current record.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiofend( )	End current format processing
fioerror( )	Perform error processing
fiofinp( )	Next input format specification

6.143 FIORNDV: Round Value

Synopsis:

```
#include "fortran.h"                PROMULA FORTRAN function declarations

int fiorndv(dspdig,ndigit,length)
char* dspdig                        Digit string to be rounded
int ndigit                          Number of rounded digits
int length                          Length of digit string
```

Description:

Truncates and rounds a numeric string of digits. The parameter `ndigit` specifies the number of digits desired in the rounded result. The parameter `length` specifies the total number of digits now in the string. The `dspdig` string may contain only numeric characters

Return value:

The function returns the carry value from the round. If the input string consists of a sequence of "999..." such that all become rounded to zero, then the output string will contain "100..." and the function will return a value of 1; else it will return a value of 0.

See also: None

6.144 FIORNL: Process FORTRAN READ DATALIST Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
    char* nmname            Name of the variable
    void* nmvalu            Points to the variable values
    int nmtype              Binary type of the variable
    int* nmadr              Points to the variable's dimensions
} namelist

int fiornl(name,nname)
namelist* name              List of variables in this namelist
int nname                   Number of variables in namelist
```

Description:

This function reads a set of variable values from the current input file. The namelist format is identical to that which may be used in specifying values in a FORTRAN DATA statement.

Return value:

A zero if the read was successful, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiostod( )	Convert string to double
fioerror( )	Perform FORTRAN I/O error processing
fiofend( )	End formatted processing
fioralph( )	Read alphabetic information
fiospace( )	Skip white space in record
fiostoi( )	Convert string to integer
fioffld( )	Get next free-form field

6.145 FIORPATH: Read Pathname Conversion Information

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiorpath(fname,pak)
char* fname                 Name of file containing conversion information
unsigned char* pak          Conversion information table
```

Description:

Reads a file containing pathname conversion information which establishes a particular pathname translation scheme. The actual specification is contained on the file whose name is specified by the `fname` parameter. This specification describes the path and file name conventions to be used on the target platform and how these conventions are to be obtained from the

source pathname specifications. The approach taken is to describe how source pathnames are to be "translated" into target pathnames.

Syntax:

```
PATHNAMES dirchar [REPLACE "s1t1s2t2..."]
                  [LOWER | UPPER]
                  [PREFIX "tname"]
                  [EXCLUDE]
                  [TERMINATION tc]

sname(1) tname(1)
.
.
.
sname(n) tname(n)
END
```

Where:

`dirchar` is the directory separation character in the source pathname

`s1t1s2t2...` are a sequence of character pairs

`tname` is a target language pathname or pathname prefix

`sname` is a source language pathname or pathname prefix

`tc` is a pathname termination character used in the source pathname

The `dirchar` specification gives the character used to separate the pathname components in the original source codes. By default, this character is replaced by the appropriate separation character / in the target pathnames.

The required `dirchar` specification specifies the character used to separate the pathname components in the original source codes. This character is replaced by an equivalent character in the target pathnames. For example moving from PRIME FORTRAN to UNIX, a "<" character would be replaced by a "/" character. Moving from MS-DOS to UNIX would replace a "\" character with a "/" character.

The optional REPLACE parameter specifies additional characters to be replaced in the source names. As many pairs of characters as are needed may be included. The standard PRIME language description, for example, contains the following specification for this option:

```
REPLACE "$_"
```

This causes all dollar signs in the source pathnames to be replaced by underscores.

The mutually exclusive and optional UPPER, LOWER parameters specify that all alphabetic characters in pathnames should be converted to upper- or lowercase respectively. Since some system's pathnames are case sensitive, while others are not, it is important to specify one of these options. For example, though most PRIME pathnames are shown in uppercase, most transfer programs create lowercase names when transferring files to UNIX; therefore, the standard PRIME language description contains a specification of LOWER for this option.

For initial testing and use of PROMULA FORTRAN for particular small projects, the simplest approach is simply to move all source files — including the INCLUDE files — into the user's local directory. To do this PROMULA FORTRAN must be told to ignore all directory information in the source pathnames. Under this alternative all characters up to and including the last occurrence of the directory component separations character are stripped from the source pathname. This is achieved via the EXCLUDE option.

A possible alternative structure for the INCLUDE files for a UNIX implementation might be to copy all of these files into some subdirectory where they would retain the same relative structure as they had on the PRIME. The PREFIX `pname` option allows a directory specification to be added to the front of all source pathnames.

Another alternative might be to copy all include files into a single subdirectory with no additional structure. This effect can be achieved by using PREFIX in conjunction with the EXCLUDE option. All source structure would be excluded and then would be replaced by the desired target subdirectory name.

In some cases, no generic translation scheme will work. Certain names might have to be changed on an individual basis. The final list of `sname, tname` pairs achieves this end. Each pathname is first translated using the generic specifications on the PATHNAME statement itself. The resultant pathnames are compared with the `snames` in the list. If the first n characters of a pathname match the n characters of an `sname`; then those n characters are stripped and the associated `tname` is added to the front of the name.

As can be seen from the above, it will be necessary to organize the INCLUDE files in the new environment. Once that organization has been completed, the PATHNAMES component of the language specification can be used to describe that structure. No changes need be made in the FORTRAN source code INCLUDE and INSERT statements.

Return Value:

The pathname conversion information is stored in the parameter `pak` as follows:

Byte	Description of content
0	Directory separation character
1	Exclude directories from pathname flag
2	Case conversion code (0 = none, 1 = toupper, 2 = tolower)
3+	Prefix to be added to name(length,characters)
4+	Conversions list

See also:

fiocpath( )	Performs the path conversion specified here

6.146 FIORTXT: Read Next Text Record

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiortxt( )
```

Description:

Reads the next physical text record into the coded communications record for detailed processing. Both "internal" and "external" files are processed.

Return value:

None, the function is void. The effect of its processing is reflected in the various global format control variables.

See also:

fioerror( )	Perform error processing

6.147 FIORWBV: FORTRAN Rewrite Binary Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiorwbv(value,nvalue)
void* value;                Points to values being written
int nvalue;                 Number of bytes to be written
```

Description:

This function rewrites binary values to a file.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.148 FIOSHL: Shift String Left

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fioshl(s,n)
char* s                     String to be shifted
int n                       Number of places to be shifted
```

Description:

Shifts a character string left a specified number of places. The spaces removed are lost.

Return value:

None, the function is void.

See also: None

6.149 FIOSHR: Shift String Right

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fioshr(s,n,fill)
char* s                     String to be shifted
int n                       Number of places to shift
char fill                   The fill character
```

Description:

Shifts a character string right a specified number of places. The spaces thus created are set equal to the specified fill character. This function is typically used during detailed editing of displays during various numeric conversions.

Return value:

None, the function is void.

See also: None

6.150 FIOSPACE: Skip White Space in Record

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiospace( )
```

Description:

This function is a utility used by the list-directed input processing functions to skip over white space within the input record. White space consists of blanks and newlines.

Return value:

If the function reads a new record during its processing, it returns a 1; else it returns a zero.

See also:

fiortxt( ) Read next text record

6.151 FIOSTATUS: Set FORTRAN I/O Error Status

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiostatus(iostat,error)
long* iostat                Address of error status variable
int error                   Error testing switch
```

Description:

If the FORTRAN I/O runtime system encounters an error, it sets an error code and calls function `fioerr`. The behavior of that function depends upon how the code using the I/O system is doing error processing. This function establishes the error code return variable and the error checking level.

Return value:

None, the function is void.

See also: None

6.152 FIOSTIO: Establish FORTRAN Standard I/O

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations
```

```
int fiostio(action)
int action                  Action code for subsequent use
```

Description:

Before any action can be performed on a standard I/O file, it must be associated with an existing FORTRAN file structure. If there is no already existing structure for the standard unit, then this function will attempt to create one.

The type of standard unit to be used is determined by the action code as follows:

Code	Standard unit
1	console (standard error)
2	standard input
3	standard printer
other	standard output

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiortxt( )	Read next text record
fioerror( )	Performs standard error processing

6.153 FIOSTOD: Convert String to Double

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

double fiostod(str,nstr)
char *str                   String to be converted
int nstr                    Length of field
```

Description:

Converts an alphanumeric string containing a number in scientific notation to a double precision floating point number. The string can contain optional leading blanks, an integer part, a fractional part, and an exponent part. The integer part consists of an optional sign followed by zero or more decimal digits. The fractional part is a decimal point followed by zero or more decimal digits. The exponent part consists of an 'E', 'e', 'D', or 'd' followed by an optional sign and a sequence of decimal digits. The parameters to this function are as follows:

Name	Description of Use
`str`	Contains the alphanumeric string to be converted.
`nstr`	Contains the number of characters in the string. Note that the string is not necessarily NULL terminated.

The following global variables are also used by this function:

Name	Description of Use
`fioerc`	If the conversion encounters a character which is not part of the notation then this variable returns the position of that character.

`fiondec`	Returns the number of decimal places in the fractional part of the number plus 1. Thus, a value of zero means there was no decimal point and one means there was a decimal, but no fractional digits.
`fioblkn`	If blanks are normal tie-breakers, then this variable is zero, else if it is +1 blanks are simply ignored and if it is -1 blanks are treated as zero.

Return value:

The double precision value of the string as computed.

See also:

fiostoi( )	Convert string to integer

6.154 FIOSTOI: Convert String to Integer

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`int fiostoi(s)`	
`char** s`	Pointer to the string pointer

Description:

Converts an alphanumeric string to an integer value. Its parameter points the location of a pointer to the start of the string. This location is updated to point immediately beyond the last character of the integer value.

Return value:

The converted value.

See also: None

6.155 FIOUWL: Establish FORTRAN Unformatted Write Length

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`void fiouwl(recl)`	
`long* recl`	Record number desired

Description:

This function merely records the length of the following unformatted record to be written.

Return value:

None, the function is void.

See also: None

6.156 FIOVFINI: Initialize A Variable FORTRAN Format

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations

```
int fiovfini(fmt,nfmt)
char* fmt;                  Pointer to the format
int nfmt;                   Length of FORMAT
```

Description:

This function initializes the variable's format strings for use in the FORMAT environment needed by the FORTRAN style input/output statements. This function is needed to convert the format string into external display code which is used by the format system.

Return value:

A zero if the format appears well-formed, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiofini( )	Initialize FORTRAN format processing

6.157 FIOWALPH: Write Alphabetic Information

Synopsis:

```
#include "fortran.h"                PROMULA FORTRAN function declarations

void fiowalph(alpha,nalpha,nfield)
char* alpha                         Character string
int nalpha                          Length of character string
int nfield                          Width of field
```

Description:

Write a character string, left-justified, blank-filled to the right into a fixed width field. If the character string is longer than the field, then only the left-most characters are written.

Return value:

None, the function is void. The effect of its processing is reflected in the various global format control variables.

See also:

fiowhexo( )	Write Hexadecimal or octal constant

6.158 FIOWBIV: FORTRAN Write Binary Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowbiv(value,nvalue)
void* value                 Points to values being written
int nvalue                  Number of bytes to be written
```

Description:

This function writes binary values to a file.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.159 FIOWDBL: Write Double Precision Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiowdbl(value,nsigdig)
double value                Value to be converted
int nsigdig                 Number of significant digits
```

Description:

Converts a double precision value to free-form display form and stores it at the current position in the coded communications record.

The parameter `value` contains the value to be converted; while `nsigdig` specifies the number of significant digits in the value.

Return value:

None, the function is void. The global variable `fiocrec` is updated to contain the new field, and the variable `fionchar` is updated to reflect the new character count in the coded communication record.

See also:

fioitos( )	Convert integer to string
fiorndv( )	Round floating point value display
fioshr( )	Shift display string right

6.160 FIOWEF: FORTRAN Write End-of-file

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowef( )
```

Description:

This function writes an end-of-file to a file.

Return value:

A zero if all went well, else an error code. See the general discussion of the FORTRAN I/O capabilities for a listing of the possible error codes.

See also: None

6.161 FIOWHEXO: Write Hexadecimal or Octal Constant

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

char* fiowhexo(base,ival,vlen)
int base;                       Base of the exact representation constant
char* ival;                     Value of exact representation constant
int vlen;                       Length of value in bytes
```

Description:

Displays a numeric value as a hexadecimal or octal character string.

Return value:

The function returns a pointer to the exact representation string for the value.

See also:

fifibit( )	Insert a bit
fifrbit( )	Reverse bits byte order
fifxbit( )	Extract a bit

6.162 FIOWLN: Write FORTRAN End-of-Line

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowln( )
```

Description:

Completes the current read operation by flushing the current format statement and writing out the current record.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiofout( )	Process output format specification
fiowtxt( )	Write text record
fiofend( )	End current format
fioerror( )	Do requested error processing

6.163 FIOWNL: Process FORTRAN WRITE DATALIST Statement

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

typedef struct {
    char* nmname;
    char* nmvalu;
    int nmtype;
    int* nmadr;
} namelist;

int fiownl(name,nname,nlident)
namelist* name;                 NAMELIST group
int nname;                      Number of variables in group
char* nlident;                  Identifier of NAMELIST group
```

Description:

Writes the information in a namelist data group.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowdbl( )	Write double precision values
fioerror( )	Perform FORTRAN I/O error processing
fiowtxt( )	Write current text record
fiodtos( )	Convert floating point number
fioitos( )	Convert integer to string

6.164 FIOWRB: Write FORTRAN Boolean Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowrb(bool,nval)
unsigned short* bool        Points to values being written
int nval                    Number of values to be written
```

Description:

Writes a vector of Boolean values to the current output file in accordance with the current format specification. In this context, the term "Boolean value" refers to a short logical value. The output display for a logical value consists of a sequence of "fw-1" blanks followed by a "T" or an "F", where "fw" is the field width. "T" is used for nonzero values and "F" is used for zero values.

Note that this function also supports the FORTRAN 66 convention under which alphabetic information may be stored in the logical values.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowtxt( )	Write a text record

6.165 FIOWRC: Write FORTRAN Character Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowrc(c,nval)
char* c                     Points to characters to be written
int nval                    Number of characters to be written
```

Description:

Writes a vector of character values to the current output file in accordance with the current format specification. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiowrih( )	Perform short integer output

6.166 FIOWRD: Write FORTRAN Double Precision Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowrd(value,nval)
double* value               Points to values being written
int nval                    Number of values to be written
```

Description:

Writes a vector of double precision values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowval( )	Write value

6.167 FIOWRF: Write FORTRAN Single Precision Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowrf(value,nval)
float* value                Points to values being written
int nval                    Number of values to be written
```

Description:

Writes a vector of single precision values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowval( )	Write value

6.168 FIOWRI: Write FORTRAN Short Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowri(value,nval)
short* value                Points to values to be written
int nval                    Number of values to be written
```

Description:

Writes a vector of short integer values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification, if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioltos( )	Convert long to string
fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowtxt( )	Write a text record

6.169 FIOWRL: Write FORTRAN Long Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowrl(value,nval)
long* value                 Points to values to be written
int nval                    Number of values to be written
```

Description:

Writes a vector of long integer values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification, if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioltos( )	Convert long to string
fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowtxt( )	Write a text record

6.170 FIOWRS: Write FORTRAN Vector of Strings

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

int fiowrs(str,nstring,nval)
char* str                       Points to start of strings
int nstring                     Length of each string
int nval                        Number of strings to be written
```

Description:

Writes a sequence of fixed length strings, stored one after another, to the current output file in accordance with the current format specification. Each string is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowtxt( )	Write a text record

6.171 FIOWRT: Write FORTRAN Truth Value Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiowrt(bool,nval)
unsigned long* bool
int nval
```
Points to values to be written
Number of values to be written

Description:

Writes a vector of truth values to the current output file in accordance with the current format specification. In this context, the term "truth value" refers to a long logical value. The output display for a logical value consists of a sequence of "fw-1" blanks followed by a "T" or an "F", where "fw" is the field width. "T" is used for nonzero values and "F" is used for zero values.

Note that this function also supports the FORTRAN 66 convention under which alphabetic information may be stored in the logical values.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiofout( )	Process output format specification
fiowalph( )	Write alphabetic information
fiowtxt( )	Write a text record

6.172 FIOWRU: Write FORTRAN Unsigned Char Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int fiowru(c,nval)
unsigned char* c            Points to characters to be written
int nval                    Number of characters to be written
```

Description:

Writes a vector of unsigned character values to the current output file in accordance with the current format specification. In this implementation `unsigned char` is derived from the nonstandard FORTRAN type LOGICAL*1. Therefore, L formatting conventions are assumed. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Do requested error processing
fiowrb( )	Perform short Boolean output

6.173 FIOWRX: Write FORTRAN Complex Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

int fiowrx(value,nval)
complex* value;             Points to values being written
int nval;                   Number of values to be written
```

Description:

Writes a vector of single precision complex values to the current output file in accordance with the current format specification. Note that in this function each individual component of the complex value is assumed to have its own corresponding floating point format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrf( )	Write vector of floats

6.174 FIOWTXT: Write Text Record

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiowtxt( )
```

Description:

Writes the current text record to the current file followed by a new line. Then it sets the length of the current text record to zero.

Return value:

None, the function is void.

See also:

fioerror( ) Perform error processing

6.175 FIOWVAL: Write Floating Point Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void fiowval(value)
double value                Value to be converted
```

Description:

This utility function controls the conversion of floating point values to string form under the control of a FORMAT specification. The actual form of the output display depends upon the particular specification. For all formats the output field consists of blanks, if necessary, followed by a minus sign if the value is negative, or an optional plus sign otherwise.

For F format this is followed by a string of digits that contains a decimal point, representing the magnitude of the value. Leading zeros are not provided, except for an optional zero immediately to the left of the decimal point if the magnitude of the value is less than one. The leading zero also appears if there would otherwise be no digits in the output field.

For E format this is followed by a zero, a decimal point, the number of significant digits specified and an exponent of a specified width.

For G format, the display type depends upon whether or not all significant digits can be displayed in F format. If they can, F format is used; if they cannot, E format is used.

For B format, business formatting conventions are used. See function `fiobfout` for details.

For all display types, if the number of digits required to represent the value is less than the specified width, the unused leftmost portion of the field is filled with blanks. If it is greater than the width, asterisks are entered instead of the representation.

Return value:

None, the function is void. The global variable `fiocrec` is updated to contain the new field, and the variable `fionchar` is updated to reflect the new character count in the coded communications record.

See also:

fiobfout( ) Perform business format output
fiodtos( ) Convert floating point number

fiorndv( )	Round floating point value display
fioshr( )	Shift string right
fiowtxt( )	Write text record

6.176 FIOWVB: Write FORTRAN Boolean Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvb(val)
unsigned short val          Boolean value to be written
```

Description:

Writes a Boolean (short logical) value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrb( )	Write Boolean value vector

6.177 FIOWVC: Write FORTRAN Character Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvc(c)
char c                      Character value to be written
```

Description:

Writes a single character value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrc( )	Write character vector

6.178 FIOWVD: Write FORTRAN Double Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvd(val)
double val                  Double precision value to be written
```

Description:

Writes a double precision value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrd( )	Write double precision vector

6.179 FIOWVF: Write FORTRAN Float Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvf(val)
float val                   Single precision value to be written
```

Description:

Writes a single precision value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrf( )	Write single precision vector

6.180 FIOWVI: Write FORTRAN Short Integer Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvi(val)
```

`short val` Short value to be written

Description:

Writes a short integer value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( ) Write binary values
fiowri( ) Write short integer vector

6.181 FIOWVL: Write FORTRAN Long Integer Value

Synopsis:

`#include "fortran.h"` PROMULA FORTRAN function declarations

```
int fiowvl(val)
long val
```
Long value to be written

Description:

Writes a long integer value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( ) Write binary values
fiowrl( ) Write long integer vector

6.182 FIOWVS: Write FORTRAN String Value

Synopsis:

`#include "fortran.h"` PROMULA FORTRAN function declarations

```
int fiowvs(str,nstring)
char* str
int nstring
```
Character string to be written
Length of string

Description:

Writes a single character string to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrs( )	Write string vector

6.183 FIOWVT: Write FORTRAN Truth Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvt(val)
unsigned long val           Truth value to be written
```

Description:

Writes a truth (long logical) value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrt( )	Write truth value vector

6.184 FIOWVU: Write FORTRAN Character Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int fiowvu(c)
unsigned char c             Character value to be written
```

Description:

Writes a single unsigned character value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowru( )	Write character vector

6.185 FIOWVX: Write FORTRAN Complex Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   float cr                 The real part of the value
   float ci                 The imaginary part of the number
} complex

int fiowvx(val)
complex val;                Single precision value to be written
```

Description:

Writes a single precision complex value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrf( )	Write single precision vector

6.186 FIOWVZ: Write FORTRAN Double Complex Value

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   double cr                The real part of the value
   double ci                The imaginary part of the number
} dcomplex

int fiowvz(val)
dcomplex val;               Double precision value to be written
```

Description:

Writes a double precision complex value to the current output file in accordance with the current format specification. This function corresponds to the FORTRAN 77 convention in which values and not just l-values can be written.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowbiv( )	Write binary values
fiowrd( )	Write double precision vector

6.187 FTNADS: FORTRAN Add Strings

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

char* ftnads(s1,n1,s2,n2)
char* s1                    Pointer to first string
int n1                      Length of first string
char* s2                    Pointer to second string
int n2                      Length of second string
```

Description:

Concatenates two FORTRAN style strings into a scratch storage array and returns a pointer to that array. Calls to this function may be nested. The maximum length of the scratch storage array is set at 256.

Return value:

A pointer to the result of the concatenation.

See also: None

6.188 FTNALLOC: Allocate Dynamic Memory

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void* ftnalloc(nbyte)
long nbyte                  Number of bytes to be allocated
```

Description:

Returns a pointer to a storage area on the heap which is at least as long as the requested number of bytes. If the bytes cannot be obtained, then this function exits to the operating system.

Return value:

A pointer to the start of the storage area.

See also: None

6.189 FTNBACK: FORTRAN Backspace Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnback(lun,...)
int lun                     Logical unit number
```

Description:

Executes the FORTRAN BACKSPACE statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the backspace operation. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
0	----	Ends the list of parameters
1	long*	Points to an error return variable

Return value:

A zero if the backspace went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiointu( )	Establish an internal file
fiolun( )	Establish unit number
fioback( )	Backspace current file
fioerror( )	FORTRAN error processing function
fiostatus( )	Set I/O Error Status

6.190 FTNBLKD: FORTRAN BLOCK DATA

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void ftnblkd()
```

Description:

This function represents a dummy entry point for those FORTRAN programs which have no BLOCK DATA subprograms. It is called by the FORTRAN runtime initialization function.

Return value:

None, the function is void.

See also: None

6.191 FTNCLOSE: FORTRAN Close Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnclose(int lun,...)
```

Description:

Executes the FORTRAN CLOSE statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the close operation. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
0	----	Ends the list of parameters
1	long*	Points to an error return variable

Return value:

A zero if the close went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiointu( )	Establish an internal file
fioclose( )	close current file
fioerror( )	Perform I/O error processing
fiolun( )	Establish unit number
fiostatus( )	Set I/O Error Status

6.192 FTNCMS: FORTRAN Compare Strings

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftncms(s1,n1,s2,n2)
char* s1                    Pointer to first string
int n1                      Length of first string
char* s2                    Pointer to second string
int n2                      Length of second string
```

Description:

Compares two FORTRAN style strings to determine their lexical relationship. The comparison proceeds character by character until either the end of both strings is reached or until a specific character difference is found. If the strings are of unequal length, the shorter string is treated as though it were padded with blanks to the length of the longer string. Note that the "lexical" value of the character is obtained from the `fifichar` function. Thus, this function does not necessarily assume the display values of the host processor, but rather can assume the display code values of another processor.

Return value:

0	if no character difference is found
–n	if a character in the first string is less than the corresponding character in the second string.
+n	if a character in the first string is greater than the corresponding character in the second string.

See also:

fifichar( )	Convert character to display code

6.193 FTNFREE: Free Dynamic Memory

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations

```
void ftnfree(ptr)
void* ptr                    Start of memory to be freed.
```

Description:

Frees dynamic memory allocated by `ftnalloc`.

Return value:

None, the function is void.

See also: None

6.194 FTNINI: Initialize FORTRAN Processing

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations

```
void ftnini( )
```

Description:

This function is called as the first executable statement of any "main" function produced by the translator. It contains the declarations of the variables needed by the FORTRAN I/O system.

The parameters `argc` and `argv` are simply the command line arguments passed to the main function and then passed here. Via the dialect description file (keyword 27) the user has the option to specify the command switches explicitly. In this case, those switches are contained in the third parameter (separated by blanks).

This function scans the command line strings looking for arguments of the following form:

Argument	Description
Cnumber	Establishes the runtime conventions code: 0 = Standard conventions 1 = Prime extended character set
Inumber	Assigns the unit whose number is indicated to standard input.
Onumber	Assigns the unit whose number is indicated to standard output.
Tnumber	Assigns the unit whose number is indicated to be a terminal — i.e., reads from standard input, writes to standard output

This function does call the "BLOCK DATA" function which is supplied with this function library as a dummy routine. This function references no other functions. Though this function must initialize many global variables, it avoids making any references to the other runtime function libraries. Thus, if you are doing no FORTRAN style input/output or no virtual memory accesses, then the code associated with those components is not loaded with your program at run-time.

Return value:

None, the function is void

See also:

ftnblkd( )	BLOCK DATA subprogram

6.195 FTNLUN: Establish File for Logical Unit Number

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`#define txtfile FILE*`	The type of a text file
`txtfile ftnlun(lun)`	
`int lun`	Logical unit number of file

Description:

Before a standard C I/O function can access a FORTRAN file, it must have access to the stream pointer associated with the logical unit number. The logical unit number must be associated with an existing FORTRAN file structure. If there is no already existing structure for the unit number, then this function will attempt to create one.

Return value:

The stream pointer associated with the logical unit number.

See also:

fiolun( )	Establish FORTRAN unit number

6.196 FTNOPEN: FORTRAN Open Statement

Synopsis:

`#include "fortran.h"`	PROMULA FORTRAN function declarations
`int ftnopen(lun,...)`	
`int lun`	Logical unit number

Description:

Executes the FORTRAN OPEN statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the open function. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
0	----	Ends the list of parameters
1	long*	Points to an error return variable
2	string	Points to the file name
3	char*	Points to the status description

4	char*	Points to access type description
5	char*	Points to the form description
6	int	Defines the record length
7	char*	Points to the blanks treatment description
8	----	Indicates that the file is read only
9	----	Indicates that the file is to be shared
10	char*	Points to the record type description

Return value:

A zero if the open went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiofdata( )	Establish file data
fioerror( )	Perform FORTRAN I/O Error Processing
fiolun( )	Establish unit number
fiointu( )	Establish an internal file
fioopen( )	Open current file
fiostatus( )	Set I/O Error Status

6.197 FTNPAUSE: FORTRAN Pause Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void ftnpause(message,nmes)
char* message               Message
int nmes;
```

Description:

Executes the "pause" operation of the FORTRAN pause statement. This function assumes that some message has already been written to `stdout`. In particular, it writes the message

```
PAUSE message
```

to console and then waits for any new record from `stdin`. If `stdin` and `stdout` are redirected, then this function may not produce the desired result.

Return value:

None, this function is void.

See also: None

6.198 FTNREAD: FORTRAN Read Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnread(lun,...)
```

`int lun` Logical unit number

Description:

Executes the FORTRAN READ statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the detailed read functions. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
1	long*	Points to an error return variable
2	---	Indicates that list-directed I/O is being performed
3	char**,int	Points to a full FORMAT specification
4	char*	Points to a FORMAT string
5	int	Specifies a record number
9	---	Specifies that more operations are to be performed
10	int	Specifies that a series of operations are desired
11		Introduces a short integer value
12		Introduces a double precision value
13		Introduces a short logical value
14		Introduces a char value
15		Introduces a long value
16		Introduces a float value
17		Introduces a long logical value
18		Introduces an unsigned char value
19		Introduces a complex value
20		Introduces a string value
21		Introduces a C-string value

Return value:

A zero if the open went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O Error Processing
fiofini( )	Initialize A FORTRAN Format
fiointu( )	Establish FORTRAN Internal Unit
fiolun( )	Establish FORTRAN Unit Number
fiorbiv( )	FORTRAN Read Binary Values
fiordb( )	Read FORTRAN Boolean Vector
fiordc( )	Read FORTRAN Char Vector
fiordd( )	Read FORTRAN Double Precision Vector
fiordf( )	Read FORTRAN Floating Point Values
fiordi( )	Read FORTRAN Short Integer Vector
fiordl( )	Read FORTRAN Long Integer Vector
fiords( )	Read FORTRAN String
fiordt( )	Read FORTRAN Truth-Value Vector
fiordu( )	Read FORTRAN Unsigned Char Vector
fiordx( )	Read FORTRAN Complex Vector
fiordz( )	Read FORTRAN Double Complex Vector
fiorec( )	Position a FORTRAN File on a Record
fiorln( )	Read FORTRAN End-of-Line
fiostatus( )	Set FORTRAN I/O Error Status
fiostio( )	Establish FORTRAN Standard I/O

6.199 FTNREW: FORTRAN Rewind Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnrew(lun,...)
int lun                     Logical unit number
```

Description:

Executes the FORTRAN REWIND statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the rewind function. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
0	----	Ends the list of parameters
1	long*	Points to an error return variable

Return value:

A zero if the rewind went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O Error Processing
fiointu( )	Establish an internal file
fiolun( )	Establish unit number
fiorew( )	Rewind current file
fiostatus( )	Set I/O Error Status

6.200 FTNSAC: FORTRAN Store a Character String

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void ftnsac(s1,n1,s2,n2)
char* s1                    Pointer to receiving string
int n1                      Length of receiving string
char* s2                    Pointer to sending string
int n2                      Length of sending string
```

Description:

Stores a FORTRAN style character string into another string and then pads that string with blanks. Note that some care must be taken with overlaying copies. Since not all platforms support an overlayed copy, this is done via an explicit loop.

Return value:

None, the function is void.

See also: None

6.201 FTNSALLO: FORTRAN String Allocation

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

typedef struct {
   char* a                  Pointer to the character storage
   int n                    Length of the string
} string;

void ftnsallo(str,...)
string* str                 String to be allocated
```

Description:

This function allocates scratch storage space needed to store a string concatenation created on the fly — in a calling parameter list or I/O statement — and then copies the specified strings into it.

Return value:

None, the function is void; however, the parameter `str` is updated to contain a pointer to the start of the created string and its length. Note that if this function is unable to obtain the needed scratch memory, it exits with a message.

See also: None

6.202 FTNSCOMP: FORTRAN String Comparison

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnscomp(s1, ns1,...)
char* s1
int ns1
```

Description:

This function performs string comparisons in which at least one of the two operands represents a concatenation of smaller strings. Once any required concatenations have been performed, the comparison proceeds character by character either until the end of both strings is reached or until a specific character difference is found. If the strings are of unequal length, the shorter string is treated as though it were padded with blanks to the length of the longer string.

Return value:

 0 if no character difference is found
–n if a character in the first string is less than the corresponding character in the second string.
+n if a character in the first string is greater than the corresponding character in the second string.

See also:

ftncms( ) Compare two FORTRAN style strings

6.203 FTNSCOPY: FORTRAN String Copy

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void ftnscopy(s1, ns1,...)
char* s1
int ns1
```

Description:

This function copies a variable number of strings into a destination string. If at the end of that copy not all space in the destination is used, it is filled with blanks.

Return value:

None, the function is void.

See also: None

6.204 FTNSLENG: FORTRAN String Length

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnsleng(s1, ns1,...)
char* s1
int ns1
```

Description:

This function computes the length of a character string or a sequence of strings being concatenated.

Return value:

The length of the result string.

See also: None

6.205 FTNSUBS: FORTRAN Substring Evaluation

Synopsis:

```
#include "fortran.h"                PROMULA FORTRAN function declarations

void ftnssubs(subs,str,slen,ipos,lpos)
string* subs                        String specification to return substring data
char* str                           String which contains the substring
int slen                            Specified length of the string
int ipos                            Starting position of the substring
int lpos                            Ending position of the substring
```

Description:

This function creates a string structure for a substring reference in a FORTRAN program. The operation it performs is minimal; however, it is needed in functional form to avoid evaluating the value of the substring bounds more than once. Note also that there is a dialectal variant here. Some allow the maximum string bound to exceed the specified string length while others do not.

Return value:

None, the function is void; however, the parameter `subs` is updated to contain a pointer to the start of the substring and its length.

See also: None

6.206 FTNSTOP: FORTRAN Stop Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void ftnstop(message)
char* message               Message to be displayed at console, i.e., stderr
```

Description:

Executes the FORTRAN STOP statement by sending a message to the console, `stderr`, not the screen, `stdout`, and making a normal exit It should be pointed out that this is one of those conventions that reasonable people do disagree about. Changing the output file to `stdout` is easily accomplished in the source for this function via a defined variable `console`.

Return value:

None, this function does not return to the calling function.

See also: None

6.207 FTNWEF: FORTRAN End File Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnwef(lun,...)
int lun                     Logical unit number
```

Description:

Executes the FORTRAN ENDFILE statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the endfile function. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
0	----	Ends the list of parameters
1	long*	Points to an error return variable

Return value:

A zero if the endfile went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform I/O error processing
fiolun( )	Establish unit number
fiointu( )	Establish an internal file
fiowef( )	Endfile to current file
fiostatus( )	Set I/O Error Status

6.208 FTNWRIT: FORTRAN Write Statement

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int ftnwrit(lun,...)
int lun                     Logical unit number
```

Description:

Executes the FORTRAN WRITE statement when translated via a non-optimized user bias. In addition to the logical unit number, this function takes a variable number of parameters which specify the actual data to be supplied to the detailed write functions. The parameter type codes passed to this function are as follows:

Code	Parameter	Description of Use
1	long*	Points to an error return variable
2	---	Indicates that list-directed I/O is being performed
3	char**,int	Points to a full FORMAT specification
4	char*	Points to a FORMAT string
5	int	Specifies a record number
9	---	Specifies that more operations are to be performed
10	int	Specifies that a series of operations are desired
11		Introduces a short integer value
12		Introduces a double precision value
13		Introduces a short logical value
14		Introduces a char value
15		Introduces a long value
16		Introduces a float value
17		Introduces a long logical value
18		Introduces an unsigned char value
19		Introduces a complex value
20		Introduces a string value
21		Introduces a C-string value

Return value:

A zero if the write went well, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fioerror( )	Perform FORTRAN I/O Error Processing
fiofini( )	Initialize A FORTRAN Format
fiointu( )	Establish FORTRAN Internal Unit
fiolun( )	Establish FORTRAN Unit Number
fiorec( )	Position a FORTRAN File on a Record
fiostatus( )	Set FORTRAN I/O Error Status
fiostio( )	Establish FORTRAN Standard I/O
fiowln( )	Write FORTRAN End-of-Line
fiowbiv( )	Write FORTRAN Binary Values
fiowrb( )	Write FORTRAN Boolean Vector
fiowrc( )	Write FORTRAN Char Vector
fiowrd( )	Write FORTRAN Double Precision Vector
fiowrf( )	Write FORTRAN Single Precision Vector
fiowri( )	Write FORTRAN Short Integer Vector
fiowrl( )	Write FORTRAN Long Integer Vector
( )	Write FORTRAN Vector of Strings
fiowrt( )	Write FORTRAN Truth-Value Vector
fiowru( )	Write FORTRAN Unsigned Char Vector
fiowrx( )	Write FORTRAN Complex Vector
fiowvb( )	Write FORTRAN Boolean Value
fiowvc( )	Write FORTRAN Character Value
( )	Write FORTRAN Double Value
fiowvf( )	Write FORTRAN Float Value
fiowvi( )	Write FORTRAN Short Integer Value
)	Write FORTRAN Long Integer Value
fiowvs( )	Write FORTRAN String Value
fiowvt( )	Write FORTRAN Truth Value
fiowvu( )	Write FORTRAN Character Value
fiowvx( )	Write FORTRAN Complex Value
fiovfini( )	Initialize A Variable FORTRAN Format

6.209 FTNXCONS: FORTRAN Exact Representation Constant

Synopsis:

```
#include "fortran.h"                    PROMULA FORTRAN function declarations

char* ftnxcons(base,xcons,xlen,vlen)
int base;                               Base of the exact representation constant
char* xcons;                            Exact representation constant in string form
int xlen;                               Length of exact representation string
int vlen;                               Length of the resultant value
```

Description:

Octal and hexadecimal character strings are physically stored as character strings and are treated as `typeless` numeric constants. This function performs this conversion. The parameter `base` specifies the base of the exact representation constant. It may have a value of 8 or 16 only. The `xcons` parameter points to the actual string for the constant. The `xlen` parameter specifies the length of the string; while the `vlen` parameter specifies the length of the resultant value. It has a maximum setting or 32.

Return value:

A pointer to the result of the conversion. This pointer must always be cast a pointer to the type of the desired value.

See also:

fifibit( )	Insert a bit
fifrbit( )	Reverse bit byte order
fifxbit( )	Extract a bit

6.210 P77GETU: Prime FORTRAN 77 Function F77$GETU

Synopsis:

```
#include "fortran.h"        Define the platform hosting this code

short p77getu(lun)
short* lun;                 Returns logical unit number
```

Description:

Scans the currently open files to find an available logical unit number.

Return value:

The unit number.

See also: None

6.211 P77NLENA: PRIME FORTRAN 77 Subroutine NLEN$A

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

short p77nlena(mes,nmes)
char* mes;                  Message to be evaluated
int nmes;                   Length of message
```

Description:

This function computes the length of a blank filled string by finding the last nonblank character.

Return value:

The length of the string

See also: None

6.212 P77TNOUA: PRIME FORTRAN 77 Subroutine TNOUA

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void p77tnoua(mes,nmes)
```

`char* mes`	Message to be written
`int nmes`	Length of message

Description:

This function writes a message to the terminal screen with a newline.

Return value:

None, the function is void.

See also: None

6.213 PDPASSN: PDP FORTRAN Subroutine Assign

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
void pdpassn(lun,name,icnt,mode,cc)
int lun;
char* name;
int icnt;
char* mode;
char* cc;
```

Logical unit number	
Name of file	
Character count for name	
File mode information	
File carriage control information	

Description:

This function assigns a file name and characteristics to a logical unit so that it may be "defined" and/or used. It is needed because the PDP dialect contains no "OPEN" statement as such.

Return value:

None, the function is void.

See also:

fiofdata( )	Establish file data
fiolun( )	Establish FORTRAN unit number
fioshl( )	Shift string left

6.214 PDPCLOSE: PDP FORTRAN Subroutine Close

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
void pdpclose(lun)
int lun;
```
Logical unit number

Description:

This function closes the file assigned to a logical unit number. It is needed because the PDP dialect contains no `CLOSE` statement as such.

Return value:

None, the function is void.

See also:

fioclose( )	Close the file
fiolun( )	Establish FORTRAN unit number

6.215 PDPCVTIM: PDP FORTRAN External Function CVTTIM

Synopsis:

```
#include "fortran.h"              PROMULA FORTRAN function declarations

void pdcvtim(midn,hrs,min,sec,tic)
long* midn;                       Number of ticks since midnight
short* hrs;                       Returns hours value
short* min;                       Returns minutes value
short* sec;                       Returns seconds value
short* tic;                       Returns ticks value
```

Description:

Converts a value which contains the number of clock ticks since midnight for a 60-cycle clock into its corresponding hours, minutes, seconds, and ticks values. The input value was computed as follows:

```
((hour * 60 + min ) * 60 + sec) * 60 + tic.
```

Return value:

None, the function is void.

See also: None

6.216 PDPGTIM: PDP FORTRAN External Function GTIM

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void pdpgtim(tic)
long* tic                   Returns number of ticks since midnight
```

Description:

Returns the current time of day. The time is returned in a long variable and is given in terms of clock ticks past midnight, though its actual resolution is only to the nearest second. The routine assumes a 60-cycle clock; therefore, the value returned is actually only

```
((hour * 60 + min ) * 60 + sec) * 60.
```

Return value:

None, the function is void.

See also: None

6.217 VMSCLS: Close Virtual File

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void vmscls( )
```

Description:

This function scans the virtual blocks in memory and writes any that have been changed back to the virtual file. This operation ensures that the virtual file can be used as a database after execution. Then it physically closes the virtual file. The normal virtual memory algorithm only writes a block when it has changed and when its memory slot is needed.

Return value:

None, the function is void.

See also:

vmsglob( )	Manages virtual global variables
vmswvb( )	Physically write block to file

6.218 VMSDEL: Change Virtual Information

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

unsigned char* vmsdel(ioiad)
long ioiad                      Offset of information to be changed
```

Description:

Returns a pointer to the byte at a specified long linear address. This function takes no responsibility for ensuring that no block boundaries are encountered downstream from the specified byte. This function assumes that the calling function intends only to change the information. The parameter is as follows:

Name	Description of use
`ioiad`	The linear address for which a pointer is desired.

Return value:

A pointer to the byte at the requested linear address.

See also:

vmsptr( )	Obtain Virtual Byte Pointer

6.219 VMSGLOB: Virtual Global Access

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void vmsglob(iop)
int iop;
```

Description:

This function loads and saves global variables from a virtual PROMULA database as specified in the globals file. This version is a dummy only to satisfy the external references from `vmsopn` and `vmscls`.

Return value:

None, the function is void.

See also: None

6.220 VMSLOAD: Load a Virtual Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

char* vmsload(nbyte,ioiad)
long nbyte;                 Number of bytes to be allocated
long ioiad;                 Virtual address of values
```

Description:

Returns a pointer to a storage area on the heap which is at least as long as the requested number of bytes. If the bytes cannot be obtained, then this function exits to the operating system. If the bytes can be obtained, they are loaded with the information at the specified address on the virtual database.

Return value:

A pointer to the start of the storage area.

See also:

ftnalloc( )	Allocate dynamic memory
vmsvect( )	Virtual vector input or output

6.221 VMSOPN: Open A Virtual Memory File

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsopn(argc,argv)
```

`int argc`	Number of command line strings
`char** argv`	An array of pointers to the strings

Description:

Opens a virtual file for processing. This may either be an existing file or a new file. This function is called to initialize the virtual file system by any program requiring the use of the virtual file system. This function is passed to the command line strings which were passed to the main function by the operating system. This function scans those command line strings looking for two particular arguments as follows:

Argument	Description
`Vname`	Specifies the name of a file which is to be used as the virtual disk file. This may be an existing PROMULA array datafile or it may be a file to be created during the execution of this program.
`Svalue`	Specifies that the name specified by the `V` parameter is to be created during the execution of this program. The parameter `value` specifies the size of that file in bytes.
`Zvalue`	Specifies the maximum number of sheets to be allocated for virtual memory. This parameter is needed when both dynamic and virtual allocations are being made. It prevents the virtual manager from exhausting all memory before the dynamic manager has a chance to get its share. The maximum setting for this parameter is 254.

Return value:

This function returns a code indicating whether or not the open was successful. The values are as follows:

Code	Meaning
0	All went well, the virtual file is ready for use
1	The existing virtual file could not be opened
2	The size of the existing file could not be determined
3	The new virtual file could not be created
4	Sufficient memory for the block status vector could not be allocated.

See also:

vmsglob( )	Manages virtual global variables

6.222 VMSPTR: Get Virtual Byte Pointer

Synopsis:

```
#include "fortran.h"                    PROMULA FORTRAN function declarations

unsigned char* vmsptr(fd, ioiad,delta)
int fd                                  Handle of virtual file
long ioiad                              Linear address of desired byte
int delta                               Change flag
```

Description:

Returns a pointer to the byte at a specified long linear address. This function takes no responsibility for ensuring that no block boundaries are encountered downstream from the specified byte. If the calling function specifies that the byte being addressed is to be changed, then this function marks the virtual block structure containing the byte accordingly. The parameters for this function are as follows:

Name	Description of use
`ioiad`	The linear address for which a pointer is desired.
`delta`	A flag which, if non-zero, indicates that the calling function intends to change the information at the byte and/or beyond.

Return value:

A pointer to the byte at the requested linear address.

See also:

vmsrbl( )	Remove virtual block from chains
vmswvb( )	Physically write block to file

6.223 VMSRBL: Remove Virtual Block

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

void vmsrbl(vlb)
int vlb                     Block to be removed
```

Description:

This function removes the indicated block from the most and least recently used chains so that the block can become the currently most recently used block.

Return value:

None, the function is void.

See also: None

6.224 VMSRDB: Read FORTRAN Virtual Boolean Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdb(value,nval)
long value                  Virtual address of values
int nval                    The number of values to be read
```

Description:

Reads a virtual vector of Boolean (short logical) values from the current input file in accordance with the current format specification. Note that in this function each individual Boolean value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordb( )	Read FORTRAN Boolean vector
vmsdel( )	Change a virtual value

6.225 VMSRDC: Read FORTRAN Virtual Character Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdc(value,nval)
long value                  Virtual address characters to be read
int nval                    Number of characters to read
```

Description:

Reads a virtual vector of character values from the current input file in accordance with the current format specification. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordc( )	Read FORTRAN character vector
vmsdel( )	Change a virtual value

6.226 VMSRDD: Read FORTRAN Virtual Double Precision Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdd(value,nval)
long value                  Virtual address values to be read
int nval                    Number of values to be read
```

Description:

Reads a virtual vector of double precision floating point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordd( )	Read FORTRAN double precision vector
vmsdel( )	Change a virtual value

6.227 VMSRDF: Read FORTRAN Virtual Floating Point Values

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdf(value,nval)
long value                  Virtual address of values to be read
int nval                    Number of values to be read
```

Description:

Reads a virtual vector of single precision floating point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordf( )	Read FORTRAN floating point vector
vmsdel( )	Change a virtual value

6.228 VMSRDI: Read FORTRAN Virtual Short Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdi(value,nval)
long value                  Virtual address of values
int nval                    Number of values to be read
```

Description:

Reads a virtual vector of short fixed point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordi( )	Read FORTRAN short integer vector
vmsdel( )	Change a virtual value

6.229 VMSRDL: Read FORTRAN Virtual Long Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdl(value,nval)
long value                  Virtual address of values
int nval                    Number of values to be read
```

Description:

Reads a virtual vector of long fixed point values from the current input file in accordance with the current format specification. Each value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordl( )	Read FORTRAN long integer vector
vmsdel( )	Change a virtual value

6.230 VMSRDS: Read FORTRAN Virtual String

Synopsis:

```
#include "fortran.h"              PROMULA FORTRAN function declarations

int vmsrds(value,nstring,nval)
long value;                       Virtual address of start of strings
int nstring;                      Length of each string
int nval;                         Number of strings to be read
```

Description:

Reads a virtual sequence of fixed length strings, stored one after another, from the current input file in accordance with the current format specification. Each string is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiords( )	Read FORTRAN character strings
vmsdel( )	Change a virtual value

6.231 VMSRDT: Read FORTRAN Virtual Truth-Value Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdt(value,nval)
long value                  Virtual address of values to be read
int nval                    The number of values to be read
```

Description:

Reads a virtual vector of truth-values (long logical values) from the current input file in accordance with the current format specification. Note that in this function each individual truth-value is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordt( )	Read FORTRAN truth value vector
vmsdel( )	Change a virtual value

6.232 VMSRDU: Read FORTRAN Virtual Unsigned Character Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmsrdu(value,nval)
long value                  Virtual address characters to be read
int nval                    Number of characters to read
```

Description:

Reads a virtual vector of character values from the current input file in accordance with the current format specification. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted read is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiordu( )	Read FORTRAN character vector
vmsdel( )	Change a virtual value

6.233 VMSSAVE: Save a Virtual Vector

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

void vmssave(svec,nbyte,ioiad)
char* svec;                     Starting address of the information
long nbyte;                     Number of bytes to be saved
long ioiad;                     Virtual address of values
```

Description:

Saves a vector of values on the virtual database and then releases the memory being used for that vector.

Return value:

None the function is void.

See also:

ftnfree( )	Free dynamic memory
vmsvect( )	Virtual vector input or output

6.234 VMSUSE: Use Virtual Information

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

unsigned char* vmsuse(ioiad)
long ioiad                      Offset of information to be used
```

Description:

Returns a pointer to the byte at a specified long linear address. This function takes no responsibility for ensuring that no block boundaries are encountered downstream from the specified byte. This function assumes that the calling function intends only to reference the information and not to change it. The parameter is as follows:

Name	Description of use

`ioiad`	The linear address for which a pointer is desired.

Return value:

A pointer to the byte at the requested linear address.

See also:

vmsptr( )	Obtain Virtual Byte Pointer

6.235 VMSVECT: Virtual Vector Input/Output

Synopsis:

```
#include "fortran.h"                    PROMULA FORTRAN function declarations

void vmsvect(fd,vec,nbyte,ioiad,iop)
int fd                                  Handle of virtual file
char* vec                               Vector being read or written
long nbyte                              Number of bytes to be processed
long ioiad                              Virtual address of values
int iop                                 Operation code: 1 = write, 0 = read
```

Description:

Reads or rewrites a vector of bytes for a specified virtual file beginning at a specified address.

Return value:

None, the function is void.

See also:

vmsptr( )	Get virtual byte pointer

6.236 VMSWRB: Write FORTRAN Virtual Boolean Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswrb(value,nval)
long value                  Virtual address values being written
int nval                    Number of values to be written
```

Description:

Writes a virtual vector of Boolean values to the current output file in accordance with the current format specification. In this context, the term "Boolean value" refers to a short logical value. The output display for a logical value consists of a sequence of "fw-1" blanks followed by a "T" or an "F", where "fw" is the field width. " T" is used for nonzero values and "F" is used for zero values.

Note that this function also supports the FORTRAN 66 convention under which alphabetic information may be stored in the logical values.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrb( )	Write FORTRAN Boolean vector
vmsuse( )	Use a virtual value

6.237 VMSWRC: Write FORTRAN Virtual Character Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswrc(value,nval)
long value                  Virtual address of characters to be written
int nval                    Number of characters to be written
```

Description:

Writes a virtual vector of character values to the current output file in accordance with the current format specification. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrc( )	Write FORTRAN character vector
vmsuse( )	Use a virtual value

6.238 VMSWRD: Write FORTRAN Virtual Double Precision Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswrd(value,nval)
long value                  Virtual address values being written
int nval                    Number of values to be written
```

Description:

Writes a virtual vector of double precision values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrd( )	Write FORTRAN double precision vector
vmsuse( )	Use a virtual value

6.239 VMSWRF: Write FORTRAN Virtual Single Precision Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswrf(value,nval)
long value                  Virtual address of values being written
int nval                    Number of values to be written
```

Description:

Writes a virtual vector of single precision values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrf( )	Write FORTRAN single precision vector
vmsuse( )	Use a virtual value

6.240 VMSWRI: Write FORTRAN Virtual Short Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswri(value,nval)
long value                  Virtual address of values to be written
int nval                    Number of values to be written
```

Description:

Writes a virtual vector of short integer values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification, if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowri( )	Write FORTRAN short integer vector
vmsuse( )	Use a virtual value

6.241 VMSWRL: Write FORTRAN Virtual Long Integer Vector

Synopsis:

```
#include "fortran.h"        PROMULA FORTRAN function declarations

int vmswrl(value,nval)
long value                  Virtual address of values to be written
int nval                    Number of values to be written
```

Description:

Writes a virtual vector of long integer values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrl( )	Write FORTRAN long integer vector
vmsuse( )	Use a virtual value

6.242 VMSWRS: Write FORTRAN Virtual Vector of Strings

Synopsis:

```
#include "fortran.h"            PROMULA FORTRAN function declarations

int vmswrs(value,nstring,nval)
long value                      Virtual address start of strings
int nstring                     Length of each string
int nval                        Number of strings to be written
```

Description:

Writes a virtual sequence of fixed length strings, stored one after another, to the current output file in accordance with the current format specification. Each string is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrs( )	Write FORTRAN string vector
vmsuse( )	Use a virtual value

6.243 VMSWRT: Write a Virtual Long Truth Value Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int vmswrt(value,nval)
long value
int nval
```
Virtual address of values to be written
Number of values to be written

Description:

Writes a virtual vector of long truth values to the current output file in accordance with the current format specification. Note that in this function each individual value is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowrt( )	Write FORTRAN long truth vector
vmsuse( )	Use a virtual value

6.244 VMSWRU: Write FORTRAN Virtual Unsigned Character Vector

Synopsis:

```
#include "fortran.h"
```
PROMULA FORTRAN function declarations

```
int vmswru(value,nval)
long value
int nval
```
Virtual address of characters to be written
Number of characters to be written

Description:

Writes a virtual vector of character values to the current output file in accordance with the current format specification. Note that in this function each individual character is assumed to have its own corresponding format specification if a formatted write is being performed.

Return value:

A zero if there is no error flag set, else an error code. See the general discussion of FORTRAN I/O capabilities for a listing of the possible error codes.

See also:

fiowru( )	Write FORTRAN character vector
vmsuse( )	Use a virtual value

6.245 VMSWVB: Write a Virtual Block

Synopsis:

```
#include "fortran.h"          PROMULA FORTRAN function declarations

typedef struct {
   int vbdel;
   int vblru;
   int vbmru;
   int vbblk;
   int vbhan;
   char vbdat[1024];
} vmsbtyp;                    Virtual File System parameters

void vmswvb(vb)
vmsbtyp* vb;
```

Description:

Physically writes the data block associated with a virtual memory block back onto its virtual file.

Return value:

None, the function is void.

See also: None